GL3.2 Version 4.1 for AIX: Programming Concepts (POWER-based Systems only)

GL3.2 Version 4.1 for AIX: Programming Concepts (POWER-based Systems only)

First Edition (October 1994)

Before using the information in this book, read the general information in Notices.

This edition applies to the GL3.2 Version 4.1 for AIX Licensed Program and to all subsequent releases of this product until otherwise indicated in new editions.

Contents

About This Book

GL3.2 Version 4.1 for AIX: Programming Concepts (POWER-based Systems Only) provides information on the Graphics Library (GL). GL is an application programming interface (API) for performing advanced 3-D rendering, window management, and input device support. This book serves as both a tutorial and a guide; it is a programmer's source book for learning about 3-D graphics from a programmer's perspective.

Who Should Use This Book

This book is intended for programmers with C programming knowledge who want to develop 3-D applications. You should be acquainted with the C programming language enough to be able to write, compile, and link a program that prints `Hello, World!` on the screen. This book does not assume knowledge of computer graphics as a prerequisite.

How to Use This Book

In general, each chapter begins with basic information and progresses to more advanced topics. On first reading, advanced topics can be skipped.

For the subroutines themselves and for example programs not shown in this book, see *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*.

The examples given in this book and in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)* are merely examples, provided for the sole purpose of illustrating that the GL basic subroutines can be used to create extended or enhanced subroutines. The subroutines are provided ″as is″ without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of each GL subroutine is with you.

Before You Begin

Having a basic understanding of the concepts of computer graphics makes this book easier to understand. An introduction to computer graphics can be found in one of the following books:

- Foley, James D.; van Dam, Andries; Feiner, Steven K.; and Hughes, John F. *Computer Graphics: Principles and Practice*, Second Edition. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.
- Rogers, David F. *Procedural Elements for Computer Graphics*. New York: McGraw-Hill Book Company, 1985.
- Hearn, Donald; and Baker, M. Pauline. *Computer Graphics*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1986.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.

Monospace	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

The following books contain information about or related to programming graphics:

- *AIX 5L Version 5.1 System Management Guide: Operating System and Devices*
- *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*
- *AIX 5L for POWER-based Systems OpenGL 2.1 Reference Manual*
- *GL3.2 Version 4.1 for AIX: Programming Concepts (POWER-based Systems Only)*

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

- AIXwindows

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Chapter 1. Graphics Library Overview

This information on the function of the Graphics Library (GL) groups subroutines with similar functions. Note that example programs not shown in this book can be found in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*. See the following for more information on the function of GL.

- Introducing the Graphics Library
- Graphics Library Subroutines and Functional Categories

Introducing the Graphics Library

GL is a set of graphics and utility subroutines that provide high- and low-level support for graphics. If you are using this library for the first time, read the following sections first.

GL Structure and Function

Graphics primitives are expressed in 2-D or 3-D user-defined coordinates. These primitives go through the graphics pipeline, which performs matrix transformations on the coordinates, clips the coordinates to normalized coordinates, and scales the transformed, clipped coordinates to screen or window coordinates. The output of the graphics pipeline is then sent to the raster subsystem.

The raster subsystem fills in the pixels between the endpoints of the lines and the interiors of polygons; draws bit-mapped characters; and performs shading, depth-cueing, and hidden surface removal. A color value for each pixel is stored in the bitplanes. The system uses the values contained in the bitplanes to display an image on the monitor.

Understanding the Hardware Used by GL

GL runs on high-resolution color workstations. GL provides a set of graphics primitives in a combination of customized, very large scale integrated (VLSI) circuits and conventional hardware, firmware, and software.

The heart of the system is the graphics pipeline. This pipeline accepts points, vectors, polygons, characters, and curves in user-defined coordinate systems and transforms them to screen coordinates. It also provides hardware support for rotation, clipping, and scaling.

In addition to the graphics pipeline, the system consists of one or more general-purpose microprocessors, a raster subsystem, a high-resolution color monitor, a keyboard, and graphics input devices.

GL windowing subroutines are implemented on top of Enhanced X-Windows. (For more information on starting Enhanced X-Windows, see the **xinit** command.) The GL drawing subroutines are implemented by means of direct adapter access. GL applications and other XClients can run simultaneously on the same X server. GL works with any X-based window manager, including the AIXwindows window manager.

On the POWER GXT1000, the multibuffer, ancillary buffer, and overlay extensions for X must be loaded to run GL.

Hardware and Software Prerequisites for GL

There are both hardware and software prerequisites for running the GL application programming interface (API).

Of the following hardware, GL must have one as a prerequisite:

- A 3-D Color Graphics Processor adapter must be installed, with or without the 24-bit color option, and with or without the 24-bit z-buffer option.

- POWER Gt4 class machines, such as the POWER Gt4, POWER Gt4x, and POWER Gt4e adapters, must be installed.
- A 7235 POWERgraphics GTO must be installed.
- The system must be a POWERstation 730 with the Supergraphics Processor Subsystem.
- The POWER GXT1000 must be installed.

GL has the following software prerequisites:

- AIX/6000 must be installed.
- AIXwindows Environment/6000 must be installed.
- AIXwindows Environment/6000 3-D Feature must be installed.

Language Bindings

GL is available with C, FORTRAN, and Ada language bindings.

New Features, Documentation, and Programs

This release of GL includes the following new or enhanced functions:

lmdef()	Support for two-sided lighting is now available on the POWER Gt4 family of graphics adapters.
set_dither()	Suport for enabling/disabling dither on the POWER Gt4 family of graphics adapters is now provided.
glcompat()	Controls backwards compatibility modes for the overlay planes and fullscreen function on the GXT1000.

The GL documentation has been updated. In particular, note that Chapter15 contains important information about using GL with the system **fork()** and **exec()** routines and about compiling and linking GL programs.

Several new example programs can be found in **/usr/lpp/GL/examples**, including:

rqenter.c	Example showing how to write re-entrant queueing code. The example creates a pipe and then checks for input on both the pipe and on the GL queue. USR1 signals caught by the process result in input being placed on the pipe. (The shell script **gen_sig.sh** can be used to generate signals.)
xinput.c	Example of X/GL integration. Shows how to simultaneously manipulate both the GL and the X11 event queues within the same process. The example can be extended to simultaneously read other (pipe, socket, or file-based) event sources.
Xcolormap.c	Example of X/GL integration. Shows how X11 can be used to manipulate the colormap associated with a GL window.
fork_examp.c	A very simple example involving GL and the system **fork()** subroutine. Shows how the GL subsystem must be shut down before using the system **fork()** routine.
fork_examp2.c	A more sophisticated **fork()** example.
GLexec.c	A very simple example involving GL and the system **exec()** routine. Shows how the GL subsystem must be shut down before using any of the system **exec()** routines.

Graphics Library Subroutines and Functional Categories

To help you select the proper subroutines for your application, the following list comprises the functional categories for GL.

Choosing the Right GL Subroutine

Select an entry from the left column for the list of subroutines in that category.

Select an entry from the right column for conceptual information on that topic.

Drawing Subroutines

Antialiasing	Draw smooth lines or points.
Attributes	Set or return attributes.
Begin-End Style Drawing	Provide fast vertex-based drawing primitives.
Rectangles, Circles, Arcs, and Polygons	Draw primitive geometric figures.
Initialization	Initialize or terminate GL functions.
Move-Draw Style Drawing	Draw primitives by moving and connecting points.
NURBS Curves and Surfaces	Draw nonuniform rational B-spline curves and surfaces.
Pipeline Option-Setting	Control flow of primitives through the graphics pipeline.
Pixels	Bit-block transfer (blit) rectangular images.
Text Characters and Strings	Create text characters and draw text strings.
Wire Frame Curves/Surfaces	Draw wire frame curves and surface patches.

Coordinate Transformation Subroutines

Coordinate Transformation	Move, rotate, or scale drawing primitives.

Viewport and Screenmask Subroutines

Viewport and Screenmask	Create and control viewports and screenmasks.

Hidden Surface Removal Subroutines

Hidden Surface Removal	Control z-buffering and backfacing polygon removal.

Lighting Subroutines

Lighting	Define multiple materials, lights, and lighting models.

Depth-Cueing Subroutines

Depth-Cueing	Make color depend on distance to viewer.

Frame Buffer Subroutines

Animation	Employ double buffering to create animated graphics.
Attributes	Set or return attributes.
Clearing, Resetting, Starting	Initialize, terminate or configure GL functions.
Color Map and RGB Modes	Manipulate color maps or work in RGB mode.
Frame Buffer Configuration	Configure frame buffer and return information.
Writemasks and Logical Ops	Control writemasks for frame buffer.
Query	Request information about system resources.

Object (Display List) Subroutines

Object (Display List)	Create and manage graphical objects (display lists).

Picking and Selecting Subroutines

Picking and Selecting	Control picking and selecting operations.

Window and Input Control Subroutines

Cursors	Define and control cursor.
Keyboard	Manage keyboard functions.
Pop-up Menus	Create and manipulate pop-up menus.
Query	Request information about system resources.
Queue and Device	Control input queues and devices.
Textport	Create and manage a screen area for textual output.
Windows	Create and manage windows.

Enhanced X-Windows GL Subroutines

Enhanced X-Windows	Control window mapping, set window properties, create widgets.

Chapter 2. Getting Started in GL

GL provides a set of fully featured functions that support graphics without depending on other graphics systems. These functions include support for input devices, a windowing system, frame buffer configuration and control, immediate and retained mode graphics, and support for basic and advanced 3-D rendering.

GL subroutines provide basic function and are limited in scope; subroutines do not interact in a complex fashion; minimal use is made of stored state; and where possible, the subroutines access hardware functionality directly. (Note that you cannot start a GL and graPHIGS application from the same process ID.)

The following series of programs introduces you to GL functions:

- Hello World Example Program for GL
- Animation Example Program for GL
- Events Example Program for GL

To run GL programs, your system must have an adapter installed that supports GL. The GL prerequisites are discussed in Hardware and Software Prerequisites. In addition, the AIXwindows server must be actively running. If the X server is not already running, you need to start it. To do this, enter the following at the command line prompt:

```
xinit
```

Hello World Example Program

The first GL program opens a window on the screen, and prints the message `Hello, World!` inside it. Create a **hello.c** file and enter the following text:

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

main ()

{
   prefsize (200, 100);
   winopen ("HI THERE");
   color (BLACK);
   clear();
   color (GREEN);
   cmov2 (50, 50);
   charstr ("Hello, World!");
   sleep (5);
}
```

To compile and link this program, enter the following command:

```
cc hello.c -o hello -lgl
```

To run the program, enter the following at the command prompt:

```
./hello
```

Depending on the configuration of your X server (as controlled by the **.Xdefaults** and **.mwmrc** files), either the window is displayed immediately on the screen or a *rubber band* is displayed. If a rubber band is displayed, you can place it at any location; to display the window, press the left mouse button. To learn more about customizing the X server for GL applications, see Understanding Windows and Input Control.

The first line of the **hello.c** program, `#include <gl/gl.h>`, includes constant, type, and function declarations needed for all GL programs. For instance, it defines the preprocessor tokens BLACK and GREEN. This line should be included in every GL program.

The **prefsize** subroutine communicates to the window manager the suggested size of the window that is created with the **winopen** subroutine. The **prefsize** subroutine does not actually create the window; it only specifies the window size preferences. In this case, the preference is a window that is 200 pixels wide and 100 pixels high. You can also control other window properties, such as the preferred position or the window title. To learn more about creating and managing windows, see Understanding Windows and Input Control.

The **color** subroutine sets the current color. Everything you draw is is displayed in the current color until you change that color. The **color** subroutine itself does not draw anything. To learn more about setting attributes, see Setting Drawing Attributes. To learn more about setting colors and frame buffer modes (an advanced topic), see Understanding the Frame Buffer.

The **clear** subroutine clears the entire window to the current color. Because the program line immediately preceding the **clear** subroutine call sets the current color to black, the window is cleared to black.

The **cmov2** subroutine sets the current character position. Every character you draw is displayed at the current character position until you change that position. In this example, the current character position is set to 50 pixels up and 50 pixels to the right of the lower left-hand corner of the window.

The **charstr** subroutine actually draws the text `Hello, World!`. The text is drawn with the current color, which is now green. To learn more about drawing text (such as choosing fonts), see Creating Text Characters. To learn about drawing lines and polygons, see Drawing with the Graphics Library.

The **sleep** subroutine is an operating system call. In this example, the **sleep** subroutine prompts the system to do nothing for 5 seconds. After 5 seconds, the **sleep** subroutine returns and the program exits. When a GL program exits, any windows that it has created disappear. Without the sleep call, the window would be created, would flash to black, with some green text, and would then disappear immediately. You can replace the sleep call with anything that will keep the program running: for instance, an infinite loop. But when the program exits, the window definitely disappears.

Animation Example Program

The previous program demonstrated some basic concepts of GL: how to open a window, how to set attributes, and how to draw. The following example program demonstrates how to create an animated scene. When this example program executes, the `Hello, World!` text moves around in a circle. This is done by clearing and redrawing the text again and again, each time at a new location.

This action is complicated by image flicker, which occurs because the system draws each image quickly, but perceptibly; that is, you do not see each individual character being drawn, only an irregular flashing and flickering. The flashing can be mild to severe, depending on what else is happening in the system or on the screen at that time.

To avoid this flashing, *double-buffering* is used. The frame buffer is partitioned into two pieces, front and back. The front buffer contains data for the pixels that are visible. The back buffer, which also contains pixel data, is invisible, but identical to the front buffer in other respects. To get smooth animation, never draw to the front buffer; instead, limit all drawing to the back buffer. When drawing is complete, the front and back buffers are swapped, and what was previously hidden is now visible. The result is smooth, flicker-free animation. The actual, step-by-step drawing process is not visible, only the final result. The following example shows how to create an animated scene by using double-buffering:

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

main ()
{
   int i, ix, iy;
```

```
   prefsize (200, 100);
   winopen ("HI THERE");
   doublebuffer ();
   gconfig ();
  for (i=1; i<1800; i++) {
     color (BLACK);
     clear ();
     color (GREEN);
     ix = (int) 40.0 * cos (((double) i) / 20.0);
     iy = (int) 40.0 * sin (((double) i) / 20.0);      cmov2 (50+ix, 50+iy);
     charstr ("Hello, World!");
     swapbuffers ();
   }
}
```

To compile this program, enter the following at the command line prompt:

```
cc hello2.c -o hello2 -lgl -lm
```

The **-lm** flag option on the **cc** command tells the linker to link to the math library, where the **sin** and **cos** functions are located.

The operation of this program is as follows: First, a window is created exactly as before. Next, the system is told to convert this window into a double-buffered window. The program does this in two steps:

1. Alerts the system with the **doublebuffer** subroutine.
2. Sets double-buffering into operation with the **gconfig** subroutine.

The process requires two steps because there are, in fact, a number of configurations into which a window can be placed. To learn more about configuring the frame buffer, an advanced topic, see Understanding the Frame Buffer .

Next, the program goes into a loop that is repeated 1800 times. Inside this loop, we clear the screen and draw the text as before. The **sin** and **cos** subroutines are operating system calls that return the sine and cosine of an angle. They are useful for drawing circular primitives. The program uses the loop counter as an angle, and moves the current character position accordingly.

Finally, when drawing is complete, the **swapbuffers** subroutine exchanges the front and the back buffers. After looping 1800 times, the program exits and the window disappears.

To learn more about creating animated scenes, see Creating Animated Scenes.

Note: Not all adapters support double buffering. You must have one of the following:

- 3-D Color Graphics Processor adapter installed, with or without the 24-bit color option, with or without the 24-bit z-buffer option.
- POWER Gt4 or POWER Gt4x adapter installed, with or without the 48-bit option.
- 7235 POWERgraphics GTO adapter installed.
- POWER GXT1000 adapter installed.
- System must be a POWERstation 730 with the Supergraphics Processor Subsystem.

Events Example Program

The first two programs demonstrated how to draw and create animated scenes, respectively. The next program demonstrates how to obtain input and illustrates the concept of an *event loop*.

The event loop is critical to writing applications for a windowing system. An event loop allows a program to respond to events occurring in the system that are beyond the control of the application program: for example, when a user picks up a window and moves it, has a window obscured and then unobscured by other windows, or resizes a window.

At this point, return to the first program, change the sleep time to 50 seconds, then recompile and rerun the program. While it is running, pick up another window (for instance, the xclock), drop it on the `HI THERE` window, pick it up again, and remove it. Notice that the original `Hello, World!` display was destroyed. This occurred because the other window overwrote the pixel data in the `HI THERE` window, and the overwritten data was not saved (GL does not support backing store or save-under).

When the contents of a window are destroyed in this fashion, the application itself must redraw the window. GL provides an event that indicates that a window may have to be redrawn. The application must test for this event, and redraw the window whenever the event is received. The discussion after the following program explains how the testing and redrawing is done.

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

/* This subroutine draws stuff */
drawstuff (int xxx, int yyy)
{
   color (BLACK);
   clear ();
   color (GREEN);
   cmov2i (xxx, yyy);
   charstr ("Hello, World!");
   swapbuffers ();
}

main ()
{
   int ox, oy;
   int ix = 150, iy = 200;
   int update = TRUE;

   /* Create and configure window */
   prefsize (400, 400);
   winopen ("HI THERE");
   doublebuffer ();
   gconfig ();

   /* get window origin */
   getorigin (&ox, &oy);

   /* queue up input devices */
   qdevice (REDRAW);  /* window needs to be redrawn         */
   qdevice (WINQUIT); /* user selected "close" from window menu */
   qdevice (MOUSEX);  /* mouse x position, in pixels */
   qdevice (MOUSEY);  /* mouse y position, in pixels */
   qdevice (ESCKEY);  /* user pressed escape key    */
   qdevice (RIGHTMOUSE);/* user pressed right mouse button */

   /* enter event loop */
   while (TRUE) {
      long dev;
      short value;

   /* if there aren't any events, and data has changed, redraw */
   if (!qtest() & update) {
      drawstuff (ix, iy);
      update = FALSE;
   }

   /* get the next event */
   dev = qread (&value);

   /* dispatch the next event */
   switch (dev) {
      case MOUSEX:
```

```
            ix = value - ox; /* update x location */
            update = TRUE;
            break;
         case MOUSEY:
            iy = value - oy;  /* update y location */
            update = TRUE;
            break;
         case REDRAW:     /* redraw it */
            getorigin (&ox, &oy); /* get window origin */
            update = TRUE;
            break;
         case ESCKEY:     /* if user presses escape key, quit */
         case RIGHTMOUSE: /* if user presses right mouse button,
                             quit */
         case WINQUIT:     /* if "close" selected from window menu */
            exit();
         default:
            break;
         }
      }
}
```

When you run this program, you will find that the character string `Hello, World!` follows the cursor around. You can can pick up the window, move it, obscure it, and uncover it, but it will always appear correctly. The following discussion examines the operation of this complicated program in detail.

First, this program includes a new header file: **gl/device.h**. This file contains definitions and type definitions (typedefs) pertaining to GL devices. To make the program easier to read, the drawing section has been put into its own subroutine, `drawstuff`, which contains a set of GL subroutines. When you want the program to draw, call this subroutine.

The program begins as before: the first step opens a window. Next, the program obtains the window origin with the **getorigin** subroutine, which will be needed later. Then, a number of devices are queued up. These devices, the REDRAW device, the MOUSEX device, and so on, generate events and place them on a queue. The program reads events from the bottom of the queue and processes them according to what came in. The **qdevice** subroutine itself does not generate or process events, but only initializes these devices and readies them for use.

Next, the program enters the event loop. Although this looks like an infinite loop, if the user presses either the escape key or the right mouse button, or else chooses the **Close** option from the window menu, the program ends.

In the loop, the program tests to see if there are any events on the event queue. If the queue is empty and the picture needs to be redrawn, the program begins to draw at this time. If the queue is not empty, then the program reads the next event and processes it. The **qread** subroutine returns the device that generated the event and a value associated with that event.

Depending on the device, the switch statement branches to the correct code to handle the event. If, for instance, the event is a mouse-motion event, the new *x* or *y* coordinate (or both) is recorded. If the event is an Escape key press, then the program is ended. After a nonterminating event is processed, the program returns to the beginning of the event loop and processes the next event.

If the user moves the window, a REDRAW event is generated. In this case, the window origin is obtained, so that the character string can be drawn in the right place later.

To learn more about devices and events, see Controlling Queues and Devices.

Chapter 3. Drawing with Graphics Library

The GL provides several categories of drawing subroutines for creating geometric figures, curves and surfaces, and character strings. Some subroutines draw a complete figure while others are very basic and draw only a line or a point. The following sections explain the drawing subroutine categories and how each works:

- Drawing with Begin-End Style Subroutines

  These subroutines draw primitive graphics figures, points, lines, and polygons, which are described as a set of vertices.
- Drawing with Move-Draw Style Subroutines

  These subroutines draw essentially the same figures as the begin-end style subroutines, but not point-sampled polygons. These subroutines are primarily included for compatibility with existing GL programs.
- Setting Drawing Attributes

  Color, texture pattern, line style, and many others attributes affect all drawing subroutines. Unless you change an attribute before calling a drawing subroutine, the subroutine renders to the screen using the attributes currently set in the system.
- Setting Pipeline Options

  Turning optional pipeline tasks on and off speeds up the rendering process.
- Drawing Rectangles, Circles, Arcs, and Polygons

  The high-level subroutines draw rectangles, circles, arcs, and polygons with varying parameters and in either filled or unfilled styles.
- Reading and Writing Pixels

  The pixel subroutines handle reading and writing pixel data, which are nongeometric drawing tasks.
- Creating Text Characters

  The text subroutines handle text drawing, a nongeometric drawing task.
- Smoothing Jagged Lines with Antialiasing

  This section deals with smoothing jagged lines of geometric figures through a method called antialiasing.
- Drawing Wire Frame Curves and Surface Patches

  These subroutines support previous methods for drawing curves and surfaces.
- Drawing NURBS Curves and Surfaces

  These subroutines support nonuniform rational B-splines (NURBS).

Drawing with Begin-End Style Subroutines

This section on begin-end style drawing discusses the following topics:

- Begin-End Style Introduction
- Lines, Polylines, and Closed Lines
- Points
- Polygons
- Point-Sampled Polygons
- Polygonal Shading
- Triangular Meshes

List of GL Begin-End Style Subroutines

The following GL begin-end style subroutines are found in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*.

bgnclosedline	Draws closed line vertices.
bgnline	Draws vertex-based lines.
bgnpoint	Draws vertex-based points.
bgnpolygon	Draws vertex-based polygons.
bgntmesh	Draws triangle mesh vertices.
concave	Allows the system to draw concave polygons.
endclosedline	Ends a series of closed line vertices.
endline	Ends a series of vertex-based lines.
endpoint	Ends a series of vertex-based points.
endpolygon	Ends a vertex-based polygon.
endtmesh	Ends a series of triangle mesh vertices.
n3f	Specifies a normal vector for lighting calculations.
normal	Specifies a normal vector for lighting calculations (can be used for display lists).
swaptmesh	Toggles the triangle mesh register pointer.
v	Transfers a vertex to the graphics pipe.

Begin-End Style Introduction

Begin-end style drawing subroutines draw primitive graphical figures. In these subroutines, all points, lines, and polygons are described in terms of vertices (sets of coordinates that identify points in space).

- A point is described by a single vertex.
- A line segment is described by two vertices indicating its end points.
- A polygon is described by a set of three or more vertices indicating its corners.

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

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

To draw a polygon whose corners are the same five points, the code takes the form:

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

Other styles of drawing you can use are described in Drawing with Move-Draw Style Subroutines and Drawing Rectangles, Circles, Arcs, and in Polygons.

Lines, Polylines, and Closed Lines

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

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

main()
{
   Int32 vert1[2] = {100, 100}; /* lower left corner */
   Int32 vert2[2] = {100, 500}; /* upper left corner */
   Int32 vert3[2] = {500, 500}; /* upper right corner */
   Int32 vert4[2] = {500, 100}; /* lower right corner */

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

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

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

The first **v2i** subroutine call after the **bgnline** subroutine creates the first end point of the line segment. The second **v2i** subroutine call after the **bgnline** subroutine creates the end point 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 of code draw a line from (100, 500) to (500, 100), the lower-right corner to the upper-left corner.

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

Polylines

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

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

```
main()
{
   Int32 vert1[2] = {200, 200};
   Int32 vert2[2] = {200, 400};
   Int32 vert3[2] = {400, 400};
   Int32 vert4[2] = {400, 200};
   prefposition(100, 500, 100, 500);
   winopen("greensquare");
   ortho2(99.5, 500.5, 99.5, 500.5);
   color(WHITE);
   clear();
   color(GREEN);
   bgnline();
    v2i(vert1);
    v2i(vert2);
    v2i(vert3);
    v2i(vert4);
    v2i(vert1);
   endline();
   sleep(3);
}
```

Note: The first vertex, `v2i(vert1)`, is repeated to close the series of line segments.

A series of connected line segments is called a polyline. GL cannot draw polylines with more than 256 vertices. Other than the number of vertices, there are no restrictions on a polyline. The segments can cross each other, vertices can be reused, and if the vertices are defined in terms of three dimensions, you can place them anywhere within three-dimensional space. In a three-dimensional space, the vertices need not all lie in the same plane.

vertex Subroutine

The previously discussed example programs crisscrossand greensquare use only one form of the vertex subroutine: a two-dimensional version with 32-bit integer coordinates. GL contains 12 forms of vertex (**v**) subroutines. The coordinates can be short integers (16 bits), long integers (32 bits), single-precision floating-point values (32 bits), and double-precision floating-point values (64 bits). For each of these types, there is a two-dimensional version, a three-dimensional version, and a version that expects vertices expressed in homogeneous coordinates.

The vertex subroutines are illustrated in the following table.

The Vertex Subroutines			
	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**

All forms of the vertex subroutine begin with the letter v. The second character is 2, 3, or 4, indicating the number of dimensions, and the final character is **s** for short integer, **i** for long integer, **f** for single-precision floating-point, and **d** for double-precision floating-point. For example, the 2-D syntaxes are as follows:

```
void v2s(Int16 vector[2])
void v2i(Int32 vector[2])
void v2f(Float32 vector[2])
void v2d(Float64 vector[2])
```

The following example program, greensquare2, illustrates the use of some of the different vertex subroutines. It draws exactly the same picture as the previous example does, but uses different versions of the **vertex** subroutine.

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

main()
{
   Int16 vert1[3] = {200, 200, 0};
   Int32 vert2[2] = {200, 400};
   Float32 vert3[2] = {400.0, 400.0};
   Float64 vert4[3] = {400.0, 200.0, 0.0};
   prefposition(100, 500, 100, 500);
   winopen("greensquare2");
   ortho2(99.5, 500.5, 99.5, 500.5);
   color(WHITE);
   clear();
   color(GREEN);
   bgnline();
    v3s(vert1);
    v2i(vert2);
    v2f(vert3);
    v3d(vert4);
    v3s(vert1);
   endline();
   sleep(3);
}
```

The previous program illustrates two things:

- Within one geometric figure (in this case, a polyline), you can mix different kinds of vertices together. In a typical application, all the vertices tend to have the same dimension and form.
- GL treats all geometric figures as three-dimensional figures. Two-dimensional versions of the vertex subroutines are actually shorthand for an equivalent three-dimensional subroutine with the *z* coordinate set to zero.

Closed Lines

In the previous two examples, the program draws a closed polyline — a line segment connecting the last point in the polyline to the first point in the polyline. Because this is a fairly common operation, there is a pair of subroutines to do it: the **bgnclosedline** and **endclosedline** subroutines.

The following program, n-gon, draws a regular, unfilled polygon centered at the origin. Specify the number of sides for the polygon on the command line when you run the program.

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

main(argc, argv)
int argc;
char *argv[];
{
   Int32 n, i;
   float vert[2];

   if (argc != 2) {
      printf("usage: %s <number of sides>\n", argv[0]);
      exit(1);
   }
   n = atoi(argv[1]);
   if (n > 256) {
      printf("Too many sides\n");
      exit(1);
   }

   prefposition(100, 500, 100, 500);
   winopen("n-gon");
   ortho2(-1.5, 1.5, -1.5, 1.5);
   color(WHITE);
```

```
   clear();
   color(RED);
   bgnclosedline();
    for (i = 0; i < n; i = i+1) {
       vert[0] = cos(i*2.0*M_PI/n);
       vert[1] = sin(i*2.0*M_PI/n);
       v2f(vert);
    }
   endclosedline();
   sleep(3);
}
```

The four lines that begin with `if (argc != 2)` test to determine whether the number of sides was entered on the command line. In other words, if the compiled file were called `ngon`, then you should run it as: `ngon 14`, or `ngon 24`. The line `n = atoi(argv[1]);` converts the parameter from ASCII to integer `n`. The **ortho2** subroutine sets the default coordinate system up so that the coordinates displayed in the window satisfy the conditions: `-1.5 <=x, y <= 1.5`.

The purpose of the previous example program is to draw exactly one `n-gon`, so there is no real penalty for computing the coordinates of the vertices between the **bgnclosedline** and **endclosedline** subroutines. If it is necessary to draw the polygon repeatedly, the calculated vertices can be saved in an array.

Other styles of drawing you can use include Drawing with Move-Draw Style Subroutines in GL and Drawing Rectangles, Circles, Arcs, and Polygons in GL.

Points

To draw a set of unconnected points in GL, enter a set of vertices specified between the **bgnpoint** and **endpoint** subroutines. The system draws each vertex as a one-pixel point on the screen. The following example program draws a set of unconnected points arranged in a square pattern. The square is 20 pixels wide by 20 pixels high, and the points are spaced 10 pixels apart.

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

main()
{
   Int32 vert[2];
   int i, j;

   prefposition(100, 500, 100, 500);
   winopen("pointpatch");
   color(BLACK);
   clear();
   color(WHITE);
   for (i = 0; i < 20; i = i+1) {
      vert[0] = 100 + 10*i;   /* load the x coordinate */
      bgnpoint();
       for (j = 0; j < 20; j = j+1) {
          vert[1] = 100 + 10*j;   /* load the y coordinate */
          v2i(vert);         /* draw the point */
          }
         endpoint();
      }
      sleep(3);
}
```

As for the line-drawing subroutines, you can have no more than 256 vertices between calls to the **bgnpoint** and **endpoint** subroutines. Consequently, the example program cannot wrap the **bgnpoint** and **endpoint** subroutines around the loop that increments the variable `i`; if it did, it would include 400 points. The program, as written, draws 20 points at a time.

The points that are drawn by the **bgnpoint** and **endpoint** subroutines are precisely one pixel in size. This size cannot be changed. Although GL does not have any explicit bgnpolymarker or endpolymarker subroutines, there are several methods you can use to get polypoints that are larger than one pixel.

If you want polypoints in the shape of raster patterns, use the font subroutines. That is, the set of rasters to use should be associated with letters of the alphabet with the **defrasterfont** subroutine. This font is then made current with the **font** subroutine. The raster patterns, which do not have to look like letters, can be positioned and drawn with the **cmov** and **charstr** subroutines, respectively.

Nonraster polymarker primitives can be created with display lists. For instance, display list line drawings in the shape of boxes, stars, crosses, asterisks, and so forth, can be created by using the **makeobj** subroutine, followed by the drawing, followed by a **closeobj** subroutine. To draw one of these items, position with the **translate** subroutine and draw it with the **callobj** subroutine.

Other aspects of begin-end style drawing include Points, Polygons, Point-Sampled Polygons, Polygonal Shading, and Triangular Meshes.

Polygons

GL draws a polygon as a filled area on the screen. It draws polygons using the same basic syntax it uses for polylines and sets of polypoints: a list of vertex subroutines surrounded by the **bgnpolygon** and **endpolygon** subroutines. For example, the following program draws a filled 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()
{
    Int32 i;

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

As with lines and points, polygons must have fewer than 256 vertices. As it does with closed lines, the GL software connects the first and the last point. You do not need to repeat the first point. An informal definition for the procedure for generating a polygon is as follows:

1. Begin with a list of vertices.
2. Draw a line segment between each vertex and the preceding one.
3. On reaching the last vertex in the list, draw a line from that vertex to the first vertex in the list.
4. Fill the area that this line circumscribes.

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

Definition of Polygons

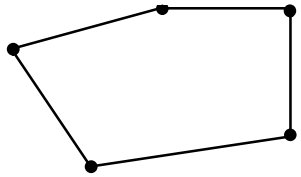
In GL, a polygon is specified by a sequence of distinct vertices, `v1, v2,..., vn`, that all lie in a plane. You can define the boundary of the polygon by connecting `v1` to `v2`, `v2` to `v3`, and so on, finally connecting `vn` back to `v1`. These connecting segments are called edges. The interior of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

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

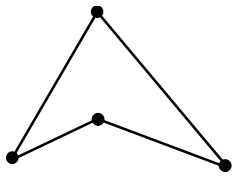
GL and the hardware can correctly render any polygon if it is simple, or if it consists of exactly four points. (Non-simple four-point polygons are often called *bowties* because of their shape.)

Some versions of the hardware automatically check for and draw concave polygons correctly, but others do not. The **concave** subroutine guarantees that the system renders concave polygons correctly. On some hardware there is a slight performance penalty when you use concave. If you intend to draw concave polygons, use the **concave** subroutine, even if your code is running on a machine that automatically does the correct thing. There is no penalty for the call, and it makes the code portable to other machines.

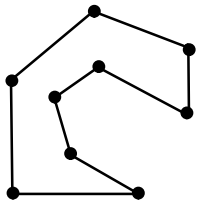
The following figures illustrate some examples of polygons. The heavy black dots represent vertices, and the lines represent edges:



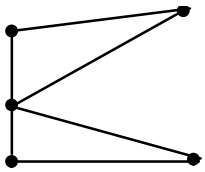
Simple Convex Polygon



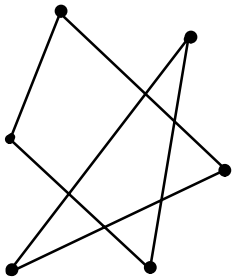
Simple Concave Polygon



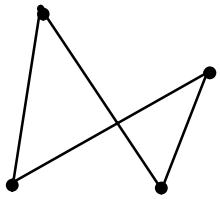
Another Simple Concave Polygon



Non-Simple Polygon



Another Non-Simple Polygon



Bow-Tie Polygon

Certain distortion problems can arise when viewing a polygon. Sometimes these distortions arise from floating-point inaccuracies. But viewing distortions can also arise if the vertices of the polygon were originally specified in three dimensions, and then were transformed and projected to two dimensions (the screen). The only distortion possible for a true polygon (that is, a polygon whose vertices lie in a single plane) is to view it edge on, in which case it collapses to a line.

However, if the defining vertices for the polygon do not all lie in a plane, the projected polygon on the two-dimensional screen might appear to have duplicate vertices, or crossing edges. Various applications may create these not-quite-true polygons when they use a mesh of polygons to model a curved surface.

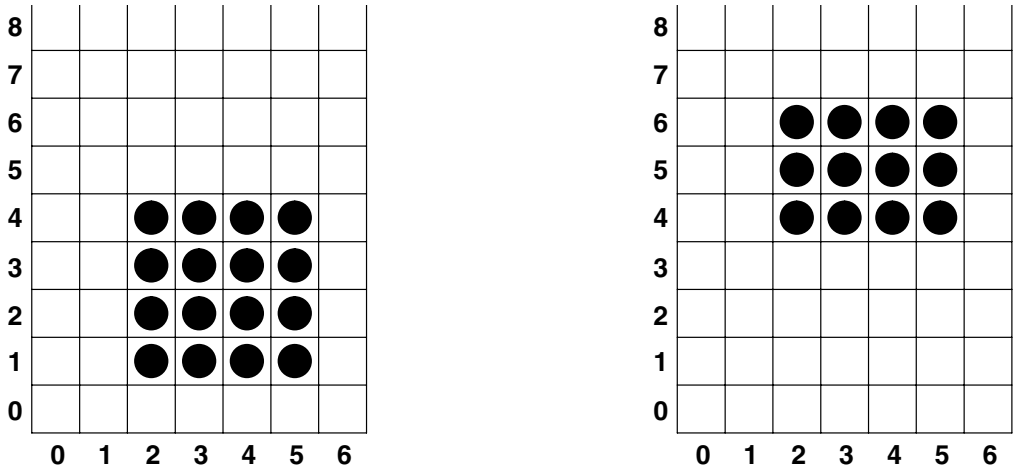
For most of the surface, the polygons formed by the mesh will be nearly flat (true) polygons. However, as the surface twists, the mesh must twist and the view of the mesh might generate bowtie polygons. This effect is most noticeable at silhouette edges where the mesh curves around to the back of the depicted object.

GL can render the bowties that arise from surface approximating meshes. In most other circumstances, however, GL subroutines for generating polygons generate only true polygons.

Point-Sampled Polygons

To represent a polygon on the screen, the system must turn on a set of pixels. Given a set of coordinates for the vertices of a polygon, there is more than one way to decide which pixels ought to be turned on. The begin-end style subroutines draw point-sampled polygons, while the move-draw style subroutines (for example, the **polf**, **rect**, and **circ** subroutines) draw outlined point-sampled polygons. The latter type of subroutine is described in Drawing with Move-Draw Style Subroutines.

To illustrate the point sampling method and the reasons for using it, consider drawing two rectangles: rectangle 1 has 2 <= *x* <= 5 and 1 <= *y* <= 4; rectangle 2 has 2 <= *x* <= 5 and 4 <= *y* <= 6. What pixels should the system turn on in both cases? The most obvious answer is shown in the figure entitled Non-Point-Sampled Polygons.

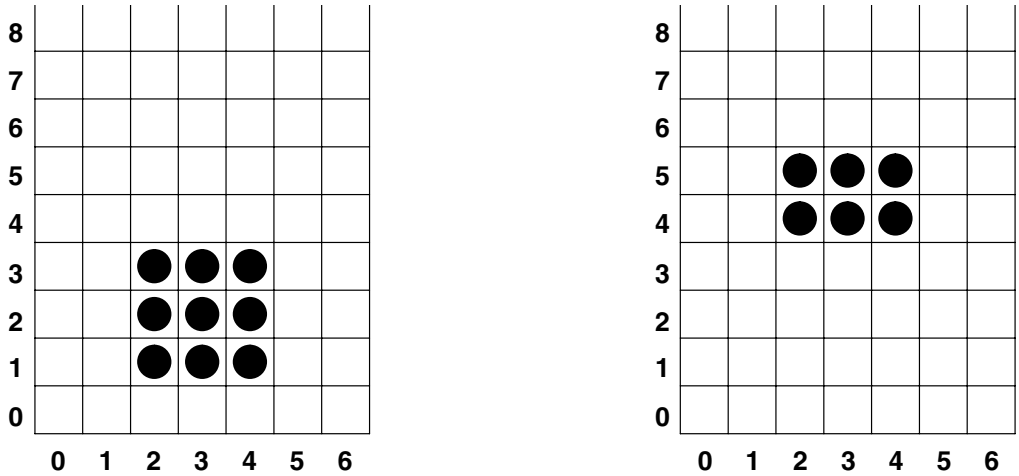


Non-Point-Sampled Polygons

If you draw a figure consisting of the two non-point-sampled polygons, you expect them to fit together. Unfortunately, if you draw them both, the pixels on the line y = 4 are drawn twice; once for each polygon. A similar problem occurs if you abut a polygon to the right. Normally, this is not a problem, but if the polygons represent a transparent surface, the duplicated edge, being twice as dense, gives the entire surface a spiderweb-like appearance.

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

GL resolves these problems by using point-sampled polygons.



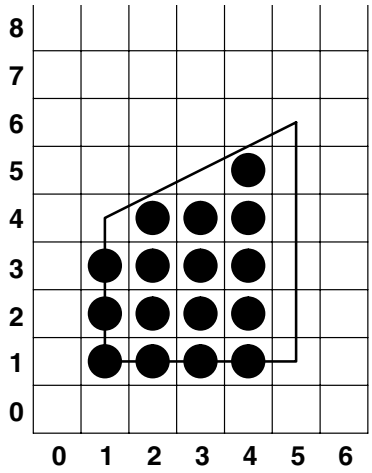
Point-Sampled Polygons

The model used assumes this: ideal mathematical lines (no thickness) connect the vertices. The system draws any pixel whose center lies inside the mathematically precise polygon. It does not draw a pixel if its center lies outside the polygon, nor any pixel whose center lies exactly on the mathematical line segments or vertices that define the polygon. The system draws the pixel only if it lies strictly within the polygon.

This definition effectively eliminates the duplication of pixels from the right and top edges of the polygon, but adjacent polygons can fill those pixels. The figure entitled Point-Sampled Polygons shows point-sampled versions of the two rectangles in Non-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 both the Non-Point-Sampled Polygons and Point-Sampled Polygons figures, the drawn areas correspond exactly to the true areas of the polygons. In nonrectangular polygons, the drawn area of the polygon cannot be exact, but the drawn area of the no-outline point-sampled polygon is closer to the true area of the polygon than the area drawn by the older outlined model.

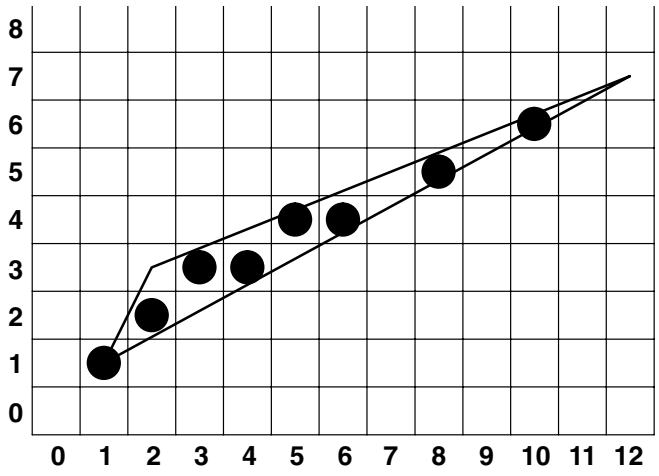
The following figure, Another Point-Sampled Polygon, illustrates the pixels that are turned on in a point-sampled representation of the polygon that connects the vertices (1,1) (1,4) (5,6) and (5,1). The darkened pixels are drawn. The pixels at (1,4), (3,5), (5,6), (5,5), (5,4), (5,3), (5,2), and (5,1) all lie mathematically on the boundary of the polygon but are not drawn because they are on the upper or right edge.



Another Point-Sampled Polygon

As mathematical entities, lines have no thickness. However, to represent a line on the screen, the system assumes a thickness of exactly one pixel. When you scale an object composed of lines, the lines behave differently from polygons. No matter how much a transformation magnifies or reduces an object composed of lines, the representation of the line remains one pixel thick. Accordingly, there is no corresponding concept of a point sampled line. If a line is drawn around a point-sampled polygon, it fills in the pixels at the upper- and right-hand edges. For compatibility, the **polf**, **rect**, and **circ** subroutines draw a line around the point-sampled version. See Drawing with Move-Draw Style Subroutines for information about these subroutines.

Anomalies can occur in the display of very thinly filled polygons. For example, consider the point-sampled rendition of the triangle connecting the points (1,1), (2,3), and (12,7). It is apparently riddled with holes, as illustrated in the following figure, Point-Sampling Anomaly. However, if adjacent polygons that share the vertices are drawn, all the pixels will eventually be filled.



Point-Sampling Anomaly

Polygonal Shading

GL offers two methods of shading polygons: flat and Gouraud. Flat-shaded polygons are those that appear flat because they are drawn with only one color. Gouraud-shaded polygons are multicolored; the interior of the polygon is a smooth blend of the colors at the vertices. Usually, the use of Gouraud shading, where appropriate, results in a significantly more realistic image.

Polygonal, or Gouraud, shading is accomplished as follows: the colors at each vertex are linearly interpolated along the edges connecting them, and then the interpolated colors on the edges are interpolated again across the interior of the polygon. The result is a smooth color variation across the entire polygon.

The interpolation is linear in all three components. For example, suppose the edge of a polygon that is 6 pixels long is colored with RGB components (0,20,100) at one end and (75,60,50) at the other. The six pixels would be colored as follows: (0,20,100), (15,28,90), (30,36,80), (45,44,70), (60,52,60), and (75,60,50).

Notice that each of the color components changes smoothly from each pixel to the next. The red component increases by 15 for each pixel, the green component increases by 8, and the blue component decreases by 10 each time. In this case, the pixel color differences work out to whole numbers. Usually this is not the case, but the approximation is done as accurately as possible.

After the colors of the pixels on the edges of the polygon are determined, the same process is used to find the colors of the pixels on the interior. The figure entitled Shaded Triangle shows the result of 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)**	**(0,12,10)**	**(0,0,0)**

Shaded Triangle

Gouraud shading also works in color map mode. In this case, the color indexes, rather than the RGB values, are interpolated. Thus, a shaded six-pixel line with endpoints colored 1 (red) and 6 (cyan) would have its six pixels colored 1, 2, 3, 4, 5, 6, or red, green, yellow, blue, magenta, cyan, assuming that the default color map is used.

Gouraud shading in color map mode is useful when false color data is being presented; for example, engineering or geophysical data such as pressure and elevation. Of course, an appropriate color ramp must be loaded to take full advantage of Gouraud shading in color map mode. A color ramp is a smooth progression of colors in the color map. For instance, the color ramp 1=red, 2=reddish-orange, 3=orange, 4=orange-yellow, 5=yellow, 6=lime-yellow would make the previous example of a six-pixel line appear smooth shaded.

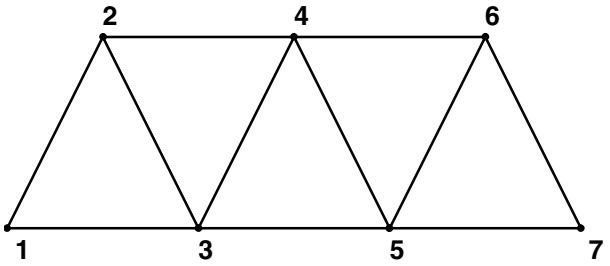
Other aspects of begin-end style drawing include Lines, Polylines, and Closed Lines; Polygons; Point-Sampled Polygons; Polygonal Shading; and Triangular Meshes.

Triangular Meshes

Triangular meshes provide a very efficient way to specify three-dimensional objects that are composed of triangular faces.

A triangular mesh is a set of triangles formed from a series of points. In the Simple Triangle Mesh figure, the seven vertices form five triangles (123, 324, 345, 546, 567). Points 1 and 7 appear in one triangle;

points 2 and 6 appear in two triangles, and all the rest appear in all three. In a longer sequence, a higher percentage of the points are used three times. If the mesh in this figure is drawn as five separate triangles, many of the points are transformed multiple times (in fact, transformation to screen coordinates occurs 15 times, although there are only 7 points). The triangular mesh primitive provides a more efficient way to display sequences of triangles.



Simple Triangle Mesh

The Simple Triangle Mesh figure illustrates the simplest case. It uses the sequence

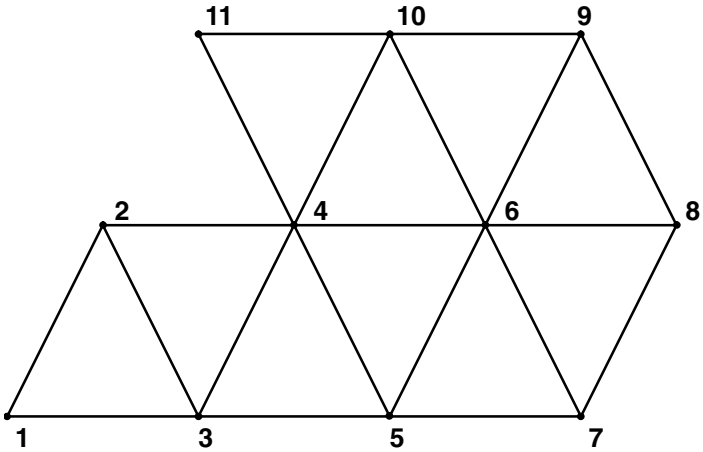
```
{bgntmesh(); v(1); v(2); v(3); v(4); v(5); v(6); v(7);
endtmesh();}
```

where `v(i)` stands for any vertex subroutine with the coordinates of the i-th point. As a result, the pipeline accepts (and transforms) points 1 and 2. When point 3 arrives, it is transformed and the system draws the triangle 123. Then point 3 replaces point 1 (so the pipeline now remembers points 2 and 3), and when point 4 arrives, triangle 324 is drawn, and point 4 replaces point 2.

This sequence continues. Each time a new point is sent, the system draws a triangle containing the new point and the two retained points. The oldest retained point is then discarded, and is replaced by the new point. The sequence ends with a call to the **endtmesh** subroutine is sent.

swaptmesh Subroutine

The Example of the swaptmesh Subroutine figure illustrates a more complex situation. The first six triangles (123, 234, 345, 456, 567, 678) could be drawn as before, but if nothing is done, the arrival of point 9 causes triangle 789 to be drawn, not triangle 689 as desired. To draw meshes like the one in this figure, we must examine more closely the mechanism the geometry hardware uses to retain points.



Example of the swaptmesh Subroutine

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

The following table illustrates what happens internally when the simple triangle mesh is drawn:

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

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

R1 = vert3

P -> R2 = vert2

The next few states are:

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

GL contains a subroutine, **swaptmesh**, whose only effect is to swap the pointer to the other retained vertex. The following sequence draws the mesh in the figure entitled Example of the swaptmesh Subroutine.

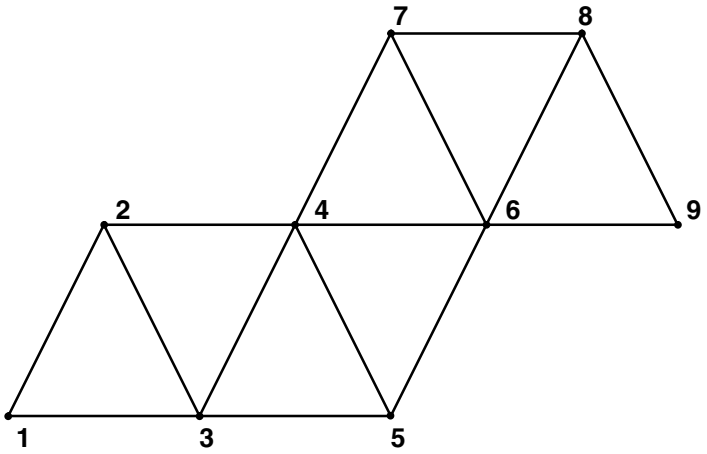
```
bgntmesh();
   v(1);
   v(2);
   v(3);
   v(4);
   v(5);
   v(6);
   v(7);
swaptmesh();
   v(8);
swaptmesh();
   v(9);
swaptmesh();
   v(10);
   v(4);
   v(11);
endtmesh();
```

The following table shows what is happening internally:

After vertex7:	**After swaptmesh:**	**After vertex8:**
R1 = vert7	P -> R1 = vert7	R1 = vert8
P -> R2 = vert6	R2 = vert6	P -> R2 = vert6
After swaptmesh:	**After vertex9:**	**After swaptmesh:**
P -> R1 = vert8	R1 = vert9	P -> R1 = vert9
R2 = vert6	P -> R2 = vert6	R2 = vert6

After vertex10:	After vertex4:	After vertex11:
R1 = vert10	P -> R1 = vert10	R1 = vert11
P -> R2 = vert6	R2 = vert4	P -> R2 = vert4

Without going into such detail, here is the sequence that draws the figure in Another swaptmesh Example:



Another swaptmesh Example

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

At most, a limit of 256 **vertex** subroutines can occur between the **bgntmesh** and **endtmesh** subroutines.

The program **octahedron.c** (in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) draws a 3-D octahedron (8-sided regular polyhedron) using the mesh primitive. Because meshes in two dimensions are of little use, the example is three-dimensional. This example uses a number of advanced concepts and routines that are covered in other sections. These include 3-D rotations, hidden surface removal, smooth (double buffered) motion, and a different color mode.

Drawing with Move-Draw Style Subroutines

This section discusses the following aspects of move-draw style subroutines:

- Move-Draw Introduction
- Current Graphics Position
- Points
- Lines and Relative Lines
- Polygons and Relative Polygons

List of GL Move-Draw Style Subroutines

The following GL Move-Draw Style Subroutines are found in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*.

draw	Draws a line.
getgpos	Gets the current graphics position.
move	Moves the current graphics position to a specified point.
pclos	Closes a filled polygon.
pdr	Specifies the next point in a polygon.
pmv	Specifies the starting point for a polygon.
pnt	Draws a point.
rdr	Draws a relative line.
rmv	Moves the current graphics position to a point relative to the current point.
rpdr	Draws a relative polygon.
rpmv	Moves the current graphics position to the starting point for a filled polygon relative to the current point.

Move-Draw Introduction

Except for polygons, the figures drawn by the move-draw subroutines are the same as those drawn by the begin-end style ones. For example, points are drawn as a single pixel. However, the subroutines draw polygons only similar to point-sampled polygons; they draw lines connecting the vertices, effectively point-sampled polygons with an outline. For many polygons, the drawing time is approximately doubled when both the polygon and its outline are drawn.

The display list subroutines described in the section entitled Working with Objects (Display Lists) (**makeobj**, **closeobj**, **callobj**, and so on) are not supported in the begin-end style subroutines.

The move-draw style subroutines used most often are the three-dimensional ones. Therefore, the naming convention assigns the shortest name (for example, the **pnt** subroutine) to the most common, 3-D form. The less-used, 2-D form is assigned the longer name, as in the **pnt2** subroutine. The 2-D versions are assumed to lie in the *z*=0 plane, but the 2-D primitives can be transformed out of that plane by the various transformation subroutines.

Current Graphics Position

In GL, the graphical figures are sent together: a set of points, a polyline, and a polygon are sent bracketed by a call to the **bgn**<*type*> and **end**<*type*> subroutines. The rendering of the figure does not start until the **end**<*type*> subroutine is received.

The system automatically maintains the current graphics position, so very few applications need to access it directly, although the **getgpos** subroutine does return the current graphics position. Its parameters include four pointers to floating point numbers in which the homogeneous coordinates of the current transformed point are returned. The form is `getgpos(&fx, &fy, &fz, &fw)`.

For compatibility, the current graphics position is maintained in exactly the same way for all the move-draw style subroutines. All other graphics subroutines do not depend on the current graphics position, and in fact, leave it in an unpredictable state.

Points

The six versions of the **pnt** subroutine are shown in the following table.

Versions of the Point Subroutine		
Parameter type	**2-D**	**3-D**
short integer	**pnt2s**	**pnts**
long integer	**pnt2i**	**pnti**
floating point	**pnt2**	**pnt**

The parameter lists are: pnt2*(x, y)* and pnt*(x, y, z)*. In addition to drawing a point, the **pnt** subroutine updates the current graphics position to its location. The syntax is as follows:

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

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

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

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

Lines and Relative Lines

Lines can be drawn using either the **move** or **draw** subroutine. Similar subroutines draw relative lines.

move and draw Subroutines

Lines can be drawn using either of two subroutines, **move** and **draw**. The syntax for the **move** and **draw** subroutines is as follows:

```
void move(Coord x, Coord y, Coord z)
void draw(Coord x, Coord y, Coord z)
```

The **move** subroutine sets the current graphics position to the specified point. The **draw** subroutine draws from the current graphics position to the specified point and then updates the current graphics position to that point. The parameters and types of the **move** and **draw** subroutines are the same as for the point subroutines. The following table is a complete list of the **move** and **draw** subroutines.

The move and draw Subroutines		
Parameter type	**2-D**	**3-D**
short integer	**move2s**	**moves**
long integer	**move2i**	**movei**
floating point	**move2**	**move**
short integer	**draw2s**	**draws**
long integer	**draw2i**	**drawi**
floating point	**draw2**	**draw**

The following example program draws the outline of a blue box on the screen using the **move** and **draw** subroutines:

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

main()
{
   prefposition(100, 500, 100, 500);
   winopen("bluebox");
   color(BLACK);
   clear();
   color(BLUE);
   move2i(200, 200);
   draw2i(200, 300);
   draw2i(300, 300);
   draw2i(300, 200);
   draw2i(200, 200);
   sleep(3);
}
```

Relative Line Subroutines

The relative line-drawing subroutines are similar to the **move** and **draw** subroutines, except that their parameters are interpreted as motions relative to the current graphics position. The relative move and draw subroutines are the **rmv** and **rdr** subroutines, respectively. After each call to a relative move or draw subroutine, the current graphics position is updated to the specified position. The syntax for the **rmv** and **rdr** subroutines is as follows:

```
void rmv(Coord x, Coord y, Coord z)
void rdr(Coord x, Coord y, Coord z)
```

If the current graphics position is (*x*, *y*, *z*), then **rdr**(*a*, *b*, *c*) draws a line from (*x*, *y*, *z*) to (*x*+*a*, *y*+*b*, *z*+*c*), and leaves the current graphics position at (*x*+*a*, *y*+*b*, *z*+*c*).

The following table is a complete list of the relative line subroutines.

Relative Line Subroutines		
Parameter type	**2-D**	**3-D**
short integer	**rmv2s**	**rmvs**
long integer	**rmv2i**	**rmvi**
floating point	**rmv2**	**rmv**
short integer	**rdr2s**	**rdrs**
long integer	**rdr2i**	**rdri**
floating point	**rdr2**	**rdr**

The following program draws a blue box identical to the blue box drawn by the previous example program. But this time, the program uses relative drawing subroutines.

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

main()
{
   prefposition(100, 500, 100, 500);
   winopen("bluebox2");
   color(BLACK);
   clear();
   color(BLUE);
   move2i(200, 200);
   rdr2i(0, 100);
   rdr2i(100, 0);
```

```
   rdr2i(0, -100);
   rdr2i(-100, 0);
   sleep(3);
}
```

Note: The first subroutine is still **move2i**; this initializes the current graphics position. If another box were to be drawn starting 200 units to the right of the first, it might begin with **rmv2i**(200, 0).

Polygons and Relative Polygons

GL provides subroutines to draw filled polygons and relative versions of filled polygons.

Filled Polygon Subroutines

The move-draw style subroutines that draw filled polygons are the **pmv** and **pdr** subroutines. The following table is a complete list of the filled polygon subroutines.

Filled Polygon Subroutines		
Parameter type	**2-D**	**3-D**
short integer	**pmv2s**	**pmvs**
long integer	**pmv2i**	**pmvi**
floating point	**pmv2**	**pmv**
short integer	**pdr2s**	**pdrs**
long integer	**pdr2i**	**pdri**
floating point	**pdr2**	**pdr**

The syntax for the **pmv** and **pdr** subroutines is as follows:

```
void pmv(Coord x, Coord y, Coord z)
void pdr(Coord x, Coord y, Coord z)
```

Relative Filled Polygon Subroutines

The relative versions of the filled polygon routines are shown in the following table:

Relative Filled Polygon Routines		
Parameter type	**2-D**	**3-D**
short integer	**rpmv2s**	**rpmvs**
long integer	**rpmv2i**	**rpmvi**
floating point	**rpmv2**	**rpmv**
short integer	**rpdr2s**	**rpdrs**
long integer	**rpdr2i**	**rpdri**
floating point	**rpdr2**	**rpdr**

The syntax for the **rpmv** and **rpdr** subroutines is as follows:

```
void rpmv(Coord x, Coord y, Coord z)
void rpdr(Coord x, Coord y, Coord z)
```

A polygon is specified by the following sequence of calls:

1. A **pmv** subroutine (or **rpmv** subroutine) to locate the first point on the boundary
2. A sequence of **pdr** (or **rpdr**) subroutines for each additional vertex
3. The **pclos** subroutine to close and fill the polygon.

The **pclos** subroutine has no parameters All the other subroutines take either two or three parameters of the appropriate type.

After any polygon subroutine, the current graphics position is left at the original point of the polygon, the point identified by the **pmv** (or **rpmv**) subroutine.

The following sample program draws a filled blue polygon:

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

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

Note: The **pclos** subroutine connects back to the original starting point.

Other drawing methods are explained in Drawing with Begin-End Style Subroutinesand Drawing Rectangles, Circles, Arcs, and Polygons.

Setting Drawing Attributes

GL contains subroutines designed to define the attributes for drawing primitives. These subroutines set the current color in either color map mode or RGB mode; the current linestyle, width, and color for line drawings; and the current fill pattern for solid objects. The **shademodel** subroutine sets the shading model for drawing shaded polygons.

You may want to read about the effects of certain attributes in Creating Lighting Effects and in Working in Color Map and RGB Modes.

To read about the effects of certain attribute subroutines, see Creating Lighting Effects in GL and Working in Color Map and RGB Modes in GL.

List of GL Drawing Attribute Subroutines

c	Sets the current color in RGB mode.
color	Sets the current color in color map mode.
cpack	Sets the current color as a packed 32-bit integer.
deflinestyle	Defines a linestyle.
defpattern	Defines a pattern.
getcolor	Returns the current color in color map mode.
getlsrepeat	Returns the linestyle repeat count.
getlstyle	Returns the current linestyle.
getlwidth	Returns the current linewidth.
getpattern	Returns the index of current fill pattern.
getsm	Returns the shading model used to draw polygons.

gRGBcolor	Returns the current color (RGB mode).
linewidth	Specifies a linewidth.
lsrepeat	Sets the repeat factor for the current linestyle.
popattributes	Pops the attribute stack.
pushattributes	Pushes down the attribute stack.
RGBcolor	Sets the current color in RGB mode.
setlinestyle	Selects a linestyle.
setpattern	Selects a pattern for filling polygons and rectangles.
shademodel	Selects a shading model used to draw polygons.

Setting Pipeline Options

Setting pipeline options is accomplished by a group of subroutines that allows a programmer to turn off and on optional tasks provided by the graphics pipeline. Turning off these tasks when they are not needed speeds the rendering process.

You may want to read about the various pipeline options in Creating Lighting Effects, Removing Hidden Surfaces, Setting Drawing Attributes, Working with Coordinate Systems, and Performing Depth-Cueing.

Various pipeline options are explained in Creating Lighting Effects in GL, Performing Depth-Cueing in GL, Removing Hidden Surfaces in GL, Setting Drawing Attributes in GL, and Working with Coordinate Systems in GL.

List of GL Pipeline Option-Setting Subroutines

backface	Allows or suppresses the display of backfacing polygons.
concave	Allows the system to draw concave polygons.
depthcue	Turns depth-cueing on or off.
getbackface	Indicates whether backfacing polygon removal is on or off.
getmmode	Returns the current matrix mode.
getsm	Returns the shading style used to draw filled polygons.
mmode	Sets the current matrix mode.
shademodel	Selects the shading style used to draw filled polygons.

Drawing Rectangles, Circles, Arcs, and Polygons

By using the move-draw style subroutines, it is possible to draw any of the primitive geometric figures in GL (except curves and surfaces). However, because these primitives are drawn so often, GL provides subroutines to draw them. The following sections explain GL high-level drawing:

- Rectangles
- Circles
- Arcs
- Polygon outlines and filled polygons

Most of the high-level subroutine names follow a pattern. If the geometric figures they draw are filled, the original subroutine name has a lowercase **f** appended to it. For example, the **rect** subroutine draws a rectangular outline, while **rectf** draws a filled (solid) rectangle. The parameters to the subroutines can be short integers (16 bits), long integers (32 bits), or floating-point numbers (32 bits). Floating point is the

default, but if the parameter type is a short integer, there is a lowercase **s** suffix. If the parameter type is a long integer, the subroutine name takes a lowercase **i** suffix. As with the **v** subroutine, only the least significant 24 bits of the long integer are considered .

List of GL Rectangle, Circle, Arc, and Polygon Subroutines

arc	Draws a circular arc.
arcf	Draws a pie-shaped filled circular arc.
circ	Draws a circle.
circf	Draws a filled circle.
polf	Draws a filled polygon.
poly	Draws a polygon.
polygonlist	Draws multiple, disjointed polygons.
polylinelist	Draws multiple, disjointed polylines.
rect	Draws a rectangle.
rectf	Draws a filled rectangle.
sbox	Draws a screen-aligned rectangle.
sboxf	Draws a filled screen-aligned rectangle.
splf	Draws a shaded filled polygon.

Rectangles

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

The following table lists the six different forms of the rectangle subroutine.

Forms of the Rectangle Subroutine		
Parameter type	**Filled**	**Unfilled**
short integer	**rectfs**	**rects**
long integer	**rectfi**	**recti**
floating point	**rectf**	**rect**

rect Subroutine

The parameters to all six versions of the **rect** rectangle subroutines are the same: rect(*x1, y1, x2, y2*). The point defined by the *x1, y1* parameters is one corner of the rectangle and that defined by the *x2, y2* parameters is the opposite corner. Because the rectangle is assumed to be aligned with the axes, the coordinates of the other corners are defined by the *x1, y2* and *y1, x2* parameters, respectively. The syntax is as follows:

```
void rect(Coord x1, Coord y1, Coord x2, Coord y2)
```

Rectangles can undergo three-dimensional geometric 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 and then viewing it in perspective. On the screen, the rotated rectangle appears to be a trapezoid.) Rectangles drawn with the **rect** subroutine (and also circles, arcs, and other 2-D figures) can be positioned anywhere in world space with the use of the routines described in Working With Coordinate Systems.

It is important to understand that the matrix manipulation routines (for instance, **translate**, **rotate**, and **scale**) execute considerably slower than the drawing subroutines such as the begin-end style subroutines. Therefore, it is almost always more efficient to draw the desired rectangle in its final position with the subroutines **bgnpolygon** and **endpolygon**, rather than using the **rect** subroutine with **translate**, **rotate**, and **scale**. Performance considerations do not necessarily apply to circles and arcs, in part because these are more complex figures and are unique in the convenience they provide.

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

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

main()
{
   Int32 i, j;

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

odd(n) /* returns 1 if n is odd; 0 otherwise. */
Int32 n;
{
   return n&1;
}
```

sbox and sboxf Subroutines

The **sbox** subroutine draws a two-dimensional, screen-aligned rectangle using the current color, writemask, linestyle, and linestyle repeat. Only these attributes, not the normal line attributes, are used. Most of the lighting/shading/viewing pipeline is bypassed.

Forms of the sbox Subroutine		
Parameter type	**Filled**	**Unfilled**
short integer	**sboxfs**	**sboxs**
long integer	**sboxfi**	**sboxi**
floating point	**sboxf**	**sbox**

The syntax is as follows:

```
void sbox(Coord x1, Coord y1, Coord x2, Coord y2)
```

The **sboxf** subroutine draws a filled rectangle.

When you use the **sbox** subroutine, you must not use lighting, backfacing, depth-cueing, z buffering, Gouraud shading, or alphablending.

Circles

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

The parameters for the circle subroutines include the center point, defined by the *x, y* parameters, and the radius. Like rectangles, circles are either filled or unfilled, and the center coordinates and radius are specified in integers, short integers, or floating-point numbers.

The following table lists the six different forms of the **circ** subroutine. The parameters to all six subroutines are the same: **circ**(*x*, *y*, *radius*).

Forms of the Circle Subroutine		
Parameter type	**Filled**	**Unfilled**
short integer	**circfs**	**circs**
long integer	**circfi**	**circi**
floating point	**circf**	**circ**

circ Subroutine

Circles are drawn with 80 equally spaced points, either as a closed line (for unfilled circles), or as a polygon (for filled circles). If your application draws many tiny circles, it is a good idea to write a circle primitive that uses fewer line segments, and which can therefore be drawn much more quickly. A similar problem can arise for very large circles. If they are magnified enough, you can see the individual straight line segments. However, circles drawn with 80 segments look smooth over a wide range of sizes. The syntax is as follows:

```
void circ(Coord x, Coord y, Coord radius)
```

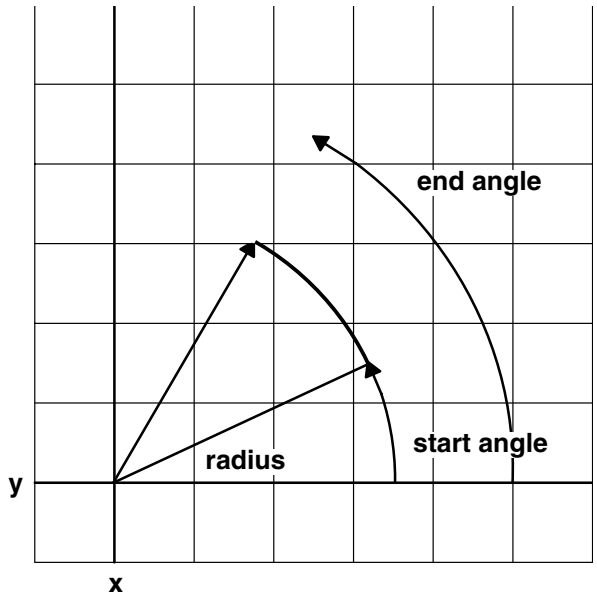
The following example program, `bullseye`, draws an archery target using filled circles:

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

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

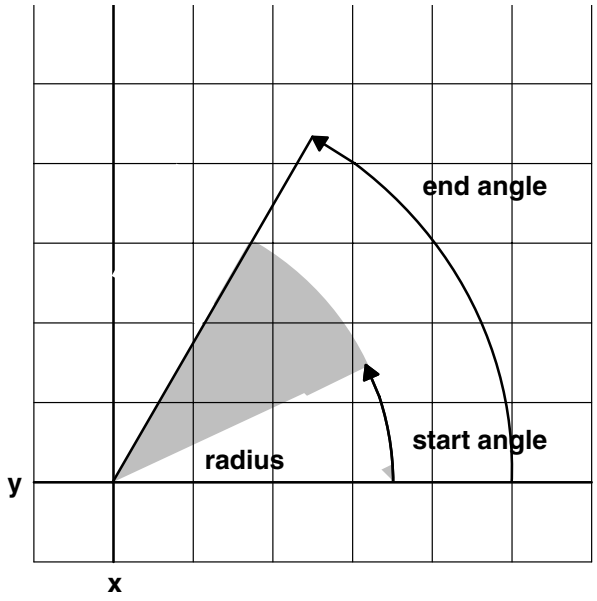
Arcs

Arcs are also two-dimensional figures, and like circles and rectangles, GL assumes they lie in the plane *z* = 0. When viewed at an angle, arcs appear to be segments of ellipses. Arcs can be either filled or unfilled. As shown in the Unfilled Arc figure, these are simply segments of circles, whereas filled arcs, shown in the Filled Arc figure, look like sections of a pie.



arc(x,y,radius,start angle, end angle);

Unfilled Arc



filled arcfi(x,y,radius,start angle,end angle);

Filled Arc

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

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

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

Arcs subroutines come in the same six forms as subroutines for circles and rectangles as shown in the following table:

Forms of the Arc Subroutine		
Parameter type	**Filled**	**Unfilled**
short integer	**arcfs**	**arcs**
long integer	**arcfi**	**arci**
floating point	**arcf**	**arc**

arc Subroutine

The parameter order for all six versions of the **arc** subroutine is *x*, *y*, *radius*, *startang*, *endang*. The syntax is as follows:

```
void arc(Coord x, Coord y, Coord radius,
          Angle startang, Angle endang)
```

The following example program, `piechart`, draws a pie chart using filled arcs:

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

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

Polygon Outlines and Filled Polygons

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

polf and poly Subroutines

Filled polygons are drawn by the **polf** subroutine, and polygon outlines are drawn by the **poly** subroutine. The following table is a complete list of the polygon and filled polygon subroutines.

Forms of the Polygon Subroutines		
Parameter type	**2-D**	**3-D**
short integer	**poly2s**	**polys**
long integer	**poly2i**	**polyi**
floating point	**poly2**	**poly**
short integer	**polf2s**	**polfs**
long integer	**polf2i**	**polfi**
floating point	**polf2**	**polf**

Both the **polf** and the **poly** subroutines take two parameters. The first parameter, *n*, is the number of vertices in the polygon, and the second, *parray*, is a two-dimensional array containing the coordinates. The syntax for the **polf** and **poly** subroutines is as follows:

```
void polf(Int32 n, Coord parray[][3])
void poly(Int32 n, Coord parray[][3])
```

This example program draws a hexagon using the **polf** subroutine:

```
#include <gl/gl.h>
Int32 parray[6][2] = {{200,100},{100,300},{200,500},
               {400,500},{500,300},{400,100}};
main()
{
   prefposition(100, 600, 100, 600);
   winopen("hexagon");
   color(BLACK);
   clear();
   color(GREEN);
   polf2i(6, parray);
   sleep(3);
}
```

Reading and Writing Pixels

Information in this section includes the following:

- Pixel Formats
- Efficient Pixel Reading and Writing
- Reading and Writing to Overlay Planes
- Other Pixel Access Subroutines

A pixel is a rectangular picture element. The display screen is composed of an array of pixels. In a black-and-white system, pixels are turned on and off to form images. In a color system, each pixel has three components: red, green, and blue. The intensity of each component can be controlled.

Pixels, like raster fonts, are not as easy to transform as geometric figures. Coding for pixel representation on the screen often requires information about the window dimensions, the screen resolution, and so forth.

Another problem with reading and writing pixels is that the contents of each pixel can mean different things depending on the display mode. The same physical bitplanes are used to store either color index information or RGB information. Accordingly, the mode of the window determines whether the contents are interpreted as RGB triples or as indexes into the color map.

The GL pixel-handling subroutines operate on arbitrarily sized rectangles, and they operate in all modes.

List of GL Pixel Block Transfer Subroutines

logicop	Specifies a logical operation for pixel writes.
pixmode	Controls the operation of the **lrectread** and **lrectwrite** subroutines.
readpixels	Returns a row of specific pixels in color map mode.
readRGB	Returns a row of specific pixels in RGB mode.
readsource	Specifies the source for pixels to be read.
rectcopy	Copies a rectangle of pixels screen to screen with optional zoom.
rectread, lrectread	Reads a rectangular array of pixels into host memory.
rectwrite, lrectwrite	Draws a rectangular array of pixels into the frame buffer.
rectzoom	Specifies the zoom factor for rectangle copies and writes.
writepixels	Paints a row of pixels on screen in color map mode.
writeRGB	Paints a row of pixels on screen in RGB mode.

Pixel Formats

The following three pixel formats constitute the standard GL pixel formats:

- Pixel data to be interpreted as red-green-blue-alpha packs 8 bits for each into a 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 red, green, blue, and alpha values are 4, 3, 2, and 1, respectively. This is exactly the same format used by the **cpack** subroutine. See Working in Color Map and RGB Modes for more information.

  Note: Alpha functions require specialized alpha hardware. The High-Performance 3-D Color Graphics Processor does not have alpha bitplanes.
- Pixels interpreted as indexes into a single, 4096-entry color map interpret the low-order 12 bits as the color index. The high-order 20 bits should be zero.
- Pixels that read or write the z buffer directly. The *z* buffer contains 24 bits of data, stored as the low-order 24 bits of a 32-bit word. The top 8 bits should be zero.

Efficient Pixel Reading and Writing

This section describes subroutines that read and write pixels with the highest possible performance.

Note: These subroutines do not check to make sure that the data is valid. If you read pixel data from a window in color map mode and then write the data into an RGB window or into the *z* buffer, the data is interpreted according to the new mode. The results are unpredictable.

pixmode Subroutine

The **pixmode** subroutine controls the format in which pixmaps are transferred to and from the adapter. You can use the **pixmode** subroutine to specify the format of a pixmap as handled by your application. The **pixmode** subroutine performs the following functions:

- Describes the format of a pixmap so that the hardware can operate with it.
- Provides a device-independent interface to pixel block transfers (BLITs).
- Allows the casual user to perform BLITs without having to understand the frame buffer organization in detail.
- Provides convenience by understanding a large variety of pixmap formats.
- Provides the most efficient interface to the hardware by minimizing unneccesary data copying.

Note: The **pixmap** subroutine is not intended to be an “image processing function” or to be used for pixmap format conversion.

After you specify the pixmap format, the **lrectread** and **lrectwrite** subroutines automatically perform the transfer from this format to the internal format of the frame buffer. The syntax is as follows:

```
void pixmode (Int32 mode, Int32 value)
```

FASTMODE is useful when it is important to store and retrieve images quickly, and when the internal format of the image is not important. A compatible mode is supplied by setting FASTMODE to FALSE. Using FASTMODE on certain adapters, for example the Supergraphics Processor Subsystem, can result in significantly improved pixel transfer capabilities.

rectread and lrectread Subroutines

The **rectread** subroutine reads a rectangular array of pixels from the window where the *xll*, *yll* parameters are the coordinates for the lower-left corner of the rectangle and the *xur*, *yur* parameters are the coordinates for the upper-right corner. All coordinates are relative to the lower-left corner of the window in screen coordinates. The syntax is as follows:

```
Int32 rectread(Screencoord xll, Screencoord yll,
Screencoord xur, Screencoord xur, Int16 *parray)
```

The *parray* parameter is an array of 16-bit values. Only the low-order 16 bits of each pixel are read, so the **rectread** subroutine is useful primarily for windows drawn in color map mode. The data is loaded into the *parray* parameter left to right, and then bottom to top. In other words, if the pixel data on the screen looked like this:

```
1  2  3  4
5  6  7  8
9 10 11 12
```

The *parray* parameter would contain `parray[0] = 9, parray[1] = 10`, or `{9, 10, 11, 12, 5, 6, 7, 8, 1, 2, 3, 4}`, and so forth. The **rectread** subroutine returns the number of pixels successfully read. Normally, this is defined as:

```
(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 the **rectread** subroutine is undefined.

Errors occur only outside the screen, not in 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 in other windows or that process images produced by other programs. The main difficulty is that the data can come from areas of the screen that are in different modes (color map mode or RGB mode). Because the **rectread** subroutine is not restricted to the current window, any or all of the coordinates can be negative.

The **lrectread** subroutine is very similar to the **rectread** subroutine. It differs in that its operation is controlled by the **pixmode** subroutine and that the *parray* parameter contains 32-bit quantities. Although the **lrectread** subroutine is useful for any kind of data, it wastes space if the data is known to be from a window in color map mode. The syntax is as follows:

```
Int32 lrectread(Screencoord xll, Screencoord yll,
                Screencoord xur, Screencoord xur,
                Int32 *parray)
```

readsource Subroutine

The **readsource** subroutine determines the source of pixels read by the **rectread**, **lrectread**, **rectcopy**, **readpixels**, and **readRGB** subroutines. The *source* parameter has four possible values defined in the **/usr/include/gl/gl.h** file. They are SRC_AUTO, SRC_FRONT, SRC_BACK, and SRC_ZBUFFER.

The default value is SRC_AUTO, which selects the front buffer in single buffer mode and the back buffer in double buffer mode. The SRC_FRONT value always reads from the front buffer (this is always valid),

and the SRC_BACK value always reads from the back buffer (valid only in double buffer mode). Finally, SRC_ZBUFFER reads 24-bit data from the z buffer. The SRC_ZBUFFER value is valid in both single and double buffer modes. The syntax is as follows:

```
void readsource(Int32 source)
```

rectwrite and lrectwrite Subroutines

The subroutines that draw rectangular arrays of pixels, **rectwrite** and **lrectwrite**, are similar to those that read pixels. The data in the *parray* parameter is 16-bit quantities for the **rectwrite** subroutine, and 32-bit quantities for the **lrectwrite** subroutine. The destination buffer is determined by the **frontbuffer**, **backbuffer**, and **zdraw** subroutines (see Double and Single Buffering and Z-Buffering). The syntax for the **rectwrite** and **lrectwrite** subroutines is as follows:

```
Int32 rectwrite(Screencoord xll, Screencoord yll,
                Screencoord xur, Screencoord xur,
                Int16 *parray)
Int32 lrectwrite(Screencoord xll, Screencoord yll,
                 Screencoord xur, Screencoord xur,
                 Int32 *parray)
```

Data is stored in the same order as in the **rectread** subroutine. In other words, if you call the **rectread** subroutine and then the **rectwrite** (or **lrectread** followed by **lrectwrite**) subroutine with the same parameters, exactly the same data is written as is read. The **rectwrite** and **lrectwrite** subroutines obey the zoom factors set by the **rectzoom** subroutine.

rectcopy Subroutine

The **rectcopy** subroutine copies the pixels from a rectangular region of the screen to a new region. As was the case with the **rectread** and **lrectread** subroutines, the source rectangle must be on the physical screen, but not necessarily constrained to the current window. The bitplane source for the pixels is determined by the **readsource** subroutine, and the destination is determined by the **frontbuffer**, **backbuffer**, and **zdraw** subroutines (see Double and Single Buffering and Z-Buffering). Self-intersecting rectangles work correctly in all cases. The syntax is as follows:

```
void rectcopy(Screencoord xll, Screencoord yll,
              Screencoord xur, Screencoord xur,
              Screencoord newx, Screencoord newy)
```

rectzoom Subroutine

With the **rectcopy** subroutine, the source rectangle can be zoomed by independent amounts in both the *x* and *y* directions. The **rectzoom** subroutine accomplishes this, where its *xfactor* and *yfactor* parameters are floating-point values defaulting to 1.0. The syntax is as follows:

```
void rectzoom(Float32 xfactor, Float32 yfactor)
```

The current system supports only integer values for the *xfactor* and *yfactor* parameters. If `rectzoom(2.0, 3.0)` is called and the following rectangle is copied:

```
1  2
3  4
```

the following copy is made:

```
1  1  2  2
1  1  2  2
1  1  2  2
3  3  4  4
3  3  4  4
3  3  4  4
```

The following program, `zoom`, magnifies the rectangular area above and to the right of the cursor to fill the window:

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

main()
{
   Int32 xsize, ysize, readxsize, readysize, x, y;
   Int32 xorg, yorg;

   winopen("zoom");
   getsize(&xsize, &ysize);
   getorigin(&xorg, &yorg);
   readxsize = xsize/3;
   readysize = ysize/3;
   rectzoom(3.0, 3.0);
   while (1) {
      x = getvaluator(MOUSEX);
      y = getvaluator(MOUSEY);
      rectcopy(x-xorg,y-yorg,x-xorg+readxsize,
               y-yorg+readysize,0,0);
   }
}
```

After determining the size and shape of the window, the program simply loops, copying a properly 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.

If you use the previous program as it 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 happens because the program is continually switching buffers, while zoom is always reading the same buffer. If you magnify the program's own window some fairly interesting effects can appear; that is, a sort of recursion takes place. These effects are enhanced if the zoom factor is set to (1.0,1.0).

Reading and Writing to Overlay Planes

To read from the overlay planes, set the pixel source with readsource (SRC_OVER). The **rectread** subroutine returns one pixel per short word; the **lrectread** subroutine returns one pixel per long word. When reading, all other bits in the short or long word are set to zero.

To write to the overlay planes, set drawmode (OVERDRAW), and then use the **rectwrite** or **lrectwrite** subroutine. Pixels must be specified one pixel per short word for the **rectwrite** subroutine and one pixel per long word for the **lrectwrite** subroutine. The pixel must lie in the lowest order bits of the short or long word. All other bits in the short or long word are ignored during a write.

To copy from one location in the overlay planes to another, specify readsource (SRC_OVER) and drawmode (OVERDRAW), and then use the **rectcopy** subroutine. The **rectcopy** subroutine does not support pixel block copies from the main frame buffer to the overlays, or vice versa. To copy to or from the main frame buffer to or from the overlays, adjust the **readsource** subroutine, use either the **rectread** or **lrectread** subroutine, adjust the **drawmode** subroutine, and then use either the **lrectwrite** or **rectwrite** subroutine.

The setting for pixmode (PM_SIZE) is ignored when performing transfers to or from the overlay planes.

Other Pixel Access Subroutines

The following GL subroutines provide other types of access to pixel blocks:

readpixels Subroutine

The **readpixels** subroutine returns values of specific pixels from the frame buffer in color map mode. It reads them into the array starting from the current character position along a single scan line (constant *y*) in the direction of increasing *x*. The syntax is as follows:

```
Int32 readpixels(Int16 number, Colorindex colors[])
```

readRGB Subroutine

The **readRGB** subroutine attempts to read specific pixel values from the frame buffer in RGB mode. The returned value of this function is the number of pixels actually read. A returned function value of 0 (zero) indicates that the starting point is not a valid character position.

> Note: The **rectread** subroutine provides significantly better performance for pixel block transfers. Even when only one row of pixels needs to be read, use the **rectread** subroutine. Do not use the **readpixels** or **readRGB** subroutines in new development.

The syntax is as follows:

```
Int32 readRGB(Int16 number, RGBvalue red[], RGBvalue green[],
              RGBvalue blue[])
```

writepixels Subroutine

The **writepixels** subroutine paints a row of pixels on the screen in color map mode.The system reads elements from the *colors* array and draws a pixel of the appropriate color for each. The syntax is as follows:

```
void writepixels(Int16 number, Colorindex colors[])
```

writeRGB Subroutine

The **writeRGB** subroutine paints a row of pixels on the screen in RGB mode. The system reads elements from the *red*, *green*, and *blue* arrays and draws a pixel of the appropriate color for each.

> Note: The **rectwrite** subroutine provides significantly better performance for pixel block transfers. Even when only one row of pixels needs to be read, use the **rectwrite** subroutine. Do not use the **writepixels** subroutine in new development.

The syntax is as follows:

```
void writeRGB(Int16 number, RGBvalue red[], RGBvalue green[],
              RGBvalue blue[])
```

Other topics affected by reading and writing pixels are Creating Text Characters and Working in Color Map and RGB Modes.

Creating Text Characters

This section includes the following aspects of character creation in GL:

- Character Strings
- International Text Support
- Fonts
- Font Query Subroutines

The GL supports the rapid display of rasterized characters in multiple fonts. The fonts can be fixed or variable pitch and can be different point sizes. You can design and use your own fonts or make use of the Enhanced X-Windows fonts supplied with the system. The system also provides query functions to determine information about the currently defined font.

For information on getting a list of available fonts, see the **XListFonts** function.

List of GL Text Subroutines

charstr	Draws a string of raster characters on the screen.
cmov	Moves the current character position.

defrasterfont	Defines bitmaps for a raster font.
font	Selects a raster font.
getcpos	Returns the current character position.
getdescender	Returns the baseline extent of the longest character descender.
getfont	Returns the current raster font number.
getfontencoding	Returns the font encoding of the current raster font.
getfonttype	Returns the font type of the current raster font.
getheight	Returns the maximum character height in the current raster font.
loadXfont	Loads an Enhanced X-Windows font into the font table.
strwidth	Returns the width of a specified text string.

Character Strings

The **cmov** subroutine determines where the system draws text on the screen, and the **charstr** subroutine draws a string of characters. The **strwidth** subroutine returns the width of a text string.

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

cmov Subroutine

The current character position determines where the system draws text on the screen. The **cmov** subroutine moves the current character position to a specified point in the same way that the **move** subroutine sets the current line-drawing position. The *x*, *y*, and *z* parameters are given as integers, short integers, or real numbers in 2-D or 3-D and specify a point in world coordinates.

The **cmov** subroutine transforms the world coordinates into window coordinates, which become the new character position. The **cmov** subroutine does not affect the current graphics position.

If the current character position is clipped out by the current viewing transformation, the character position is set to invalid, and any character strings that are drawn do not appear. The **cmov** subroutine does not cause anything to be drawn. It simply sets the current character position where drawing occurs when the **charstr** subroutine is issued.

The parameters are (*x*, *y*) for the **cmov2** subroutines and (*x*, *y*, *z*) for the **cmov** subroutines. The syntax for the **cmov** and **cmov2** subroutines is as follows:

```
void cmov(Coord x, Coord y, Coord z)
void cmov2(Coord x, Coord y)
```

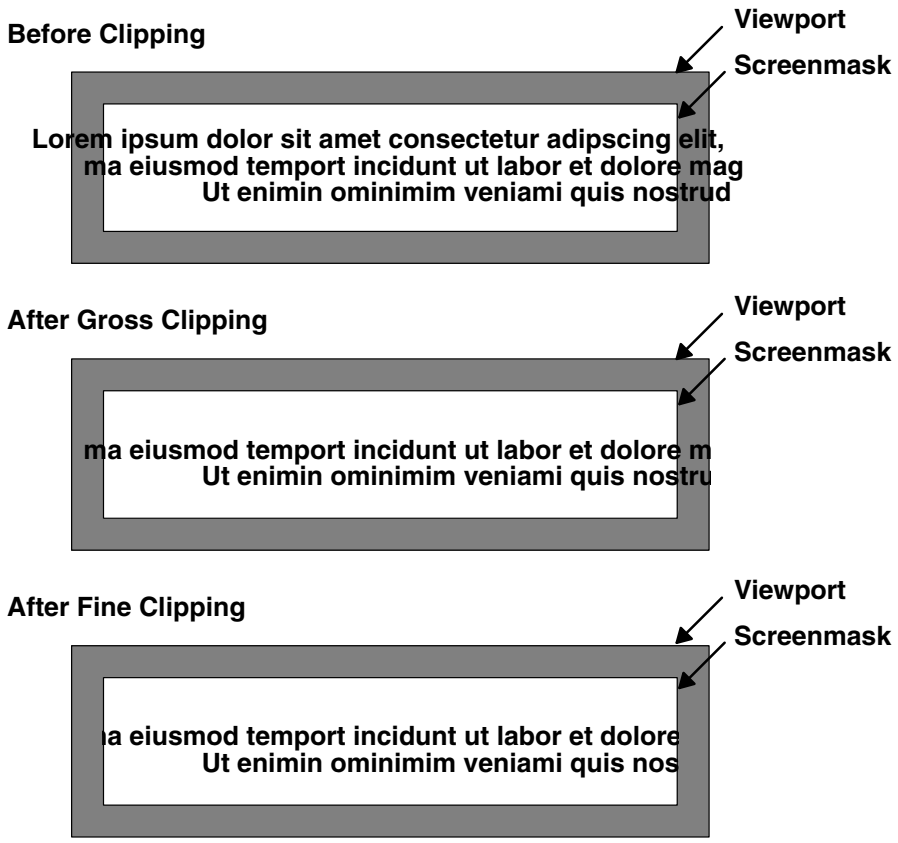
Forms of the cmov Subroutine		
Parameter Type	**2-D**	**3-D**
Int16	**cmov2s**	**cmovs**
Int32	**cmov2i**	**cmovi**
Float	**cmov2**	**cmov**

charstr Subroutine

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

```
void charstr(Char8 *string)
```

If the origin of a character string lies outside the viewport, none of the characters in the string are drawn. If the origin is inside the viewport, the characters are individually clipped to the screenmask. The screenmask is normally set to the same size as the viewport, although it can be set smaller than the viewport to enable two kinds of clipping, as illustrated in the Gross and Fine Clipping ″figure″ on page 45 .



Gross and Fine Clipping

Gross clipping removes all strings that start outside the viewport. Fine clipping trims individual characters to the screenmask. Viewport clipping and screenmask clipping apply to all drawing primitives as well. However, the difference between these two types of clipping is usually not important, except for character strings.

Characters are drawn in the current color. Unless the font is specifically changed in the program, character strings are drawn in the current font. The default font is `font0`, defined when the **winopen** subroutine is called. The following example program, `rasterchars`, draws two lines of text. The program assumes the default font is less than 12 pixels high.

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

main()
{   prefposition(100, 500, 100, 500);
   winopen("rasterchars");
   color(BLACK);
```

```
   clear();
   color(RED);
   cmov2i(50,80);
   charstr("The first line is drawn ");
   charstr("in two parts. ");
   cmov2i(50, 66);
   charstr("This line is 14 pixels lower. ");
   sleep(10); /* pause for ten seconds */
}
```

The `rasterchars` program illustrates the following:

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

> **Note:** The characters are drawn in the current color. Because nothing was mentioned in the program about fonts, all the strings are drawn in the current font.

Example Program Using the rotate Subroutine

The following example uses the **rotate** subroutine and illustrates that character strings are drawn in the same orientation no matter where they move, and that the current character move function of the **cmov** subroutine is transformed like any other geometry. In the example, the **rotate** subroutine rotates the polygon about the *z* axis (coming directly out of the screen) by 5 degrees each time. The rotation is about the origin, so vertex `p1` should remain fixed. The vertex callouts rotate with the vertices.

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

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

main()
{
   long i;

   prefposition(100, 500, 100, 500);
   winopen("rasterchars");
   ortho2(-1.0, 1.0, -1.0, 1.0);
   for (i = 0; i < 40; i++) {
      color(BLACK);

      clear();
      rotate(50, 'z');
      color(RED);
      bgnpolygon();
      v2f(p1); v2f(p2); v2f(p3);

      endpolygon();
      color(GREEN);
      cmov2(0.0, 0.0);
      charstr("vert1");
      cmov2(0.6, 0.0);

      charstr("vert2");
      cmov2(0.0, 0.6);
      charstr("vert3");
      sleep(1);
   }
}
```

International Text Support

The **charstr** subroutine supports both single-and double-byte raster character rendering.

If the current font is a double-byte font, this subroutine expects the first two bytes to represent the first character, the second two bytes to represent the next character, and so on. Double-byte fonts are useful in languages with extremely large character sets such as Japanese and Chinese.

If the current font is a single-byte font, each byte represents one character. The ASCII code set is an example of a single-byte font.

It is the user's responsibility to determine if the currently bound font is a single- or double-byte font and to pass the appropriate string. To determine the font type, use the **getfonttype** subroutine.

Using Double-Byte Character Sets

Currently, the **winopen** subroutine does not support double-byte character set (DBCS) window titles. To set the name of a GL window to a DBCS string, use the **XSetWMName** subroutine. The window id of a GL window can be obtained with the **getXwid** subroutine. To use DBCS strings in a pop-up menu, redefine the default font (font id 0) to be a DBCS font. This can be done with the **loadXfont** subroutine. Note that only **loadXfont** can be used to redefine font id 0; the **defrasterfont** subroutine does not allow font id 0 to be redefined. Be sure to specify pop-up menu entries as DBCS strings. The **charstr** subroutine does support DBCS output, provided that the current font is a DBCS font.

Using the charstr Subroutine to Render Japanese and Asian Fonts

The following code fragment demonstrates how double-byte fonts can be loaded and used for rendering in GL. An example program illustrating this use can be found in the **/usr/lpp/GL/examples/jischarstr.c** file.

```
char* jisX0201 =
 "-ibm_aix-gothic-medium-r-normal--35-230-100-100-m-170-jisx0201.1976-0";

char* jisX0208 =
 "-ibm_aix-gothic-medium-r-normal--35-230-100-100-m-340-jisx0208.1983-0";   char* ibmUDC   =
   "-ibm_aix-gothic-medium-r-normal--35-230-100-100-m-340-ibm-udcjp";   loadXfont( 2, jisX0201);   /* Kana Font */
   loadXfont( 3, jisX0208);   /* Kanji Font */
   loadXfont( 4, ibmUDC);     /* IBM User Defined Chars */

 font(2)
 charstr("k1k2k300");      /* Draw Kana Character */

 font(3);
 charstr("K1K2K300");      /* Draw Kanji string */
```

> Note: The two null bytes are required to terminate a DBCS string. A single null byte is not sufficient; a short word whose first byte is null, but whose second byte is not is still a valid glyph index.

strwidth Subroutine

The **strwidth** subroutine returns the width of a text string in pixels, using the character-spacing parameters in the current raster font. This text string can be any null-terminated ASCII string of characters. Characters in some fonts may not be all the same width, so the **strwidth** subroutine does not necessarily return the width of a character times the number of characters in the string. The syntax is as follows:

```
Int32 strwidth(Char8 *string)
```

Fonts

A GL raster font is a collection of up to 255 rectangular arrays of masks. If a 1 (one) appears in a mask, then the corresponding pixel is turned on to the current color. If a 0 (zero) appears, the pixel remains as it is. For example, the following bitmasks might be used to draw the character A:

```
Binary          Hexadecimal
0000011000000000 = 0x0600
0000011000000000 = 0x0600
0000111100000000 = 0x0F00
0000111100000000 = 0x0F00
0001100110000000 = 0x1980
0001100110000000 = 0x1980
0011000011000000 = 0x30C0
```

```
0011111111000000 = 0x3FC0
0110000001100000 = 0x6060
0110000001100000 = 0x6060
1100000000110000 = 0xC030
1100000000110000 = 0xC030
```

To define a single character, you need the following variables:

- bitmask
- width
- height
- xoffset
- yoffset
- xincrement

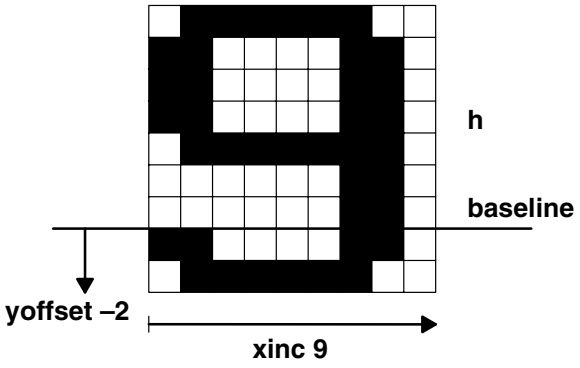
The **defrasterfont** subroutine allows you to define a collection of characters with these variables.

Raster font characters are defined by a bitmap, 1 bit per pixel. The width and height of the character, the number of bits in one row of the bitmap, and the baseline position are also specified.

Each array of bitmasks defining an ASCII character makes up a bitmask entry for that character. For the previous example, the ASCII value of A is 65 (decimal), so entry 65 in the font is associated with the bitmask. If such a font were defined, the string AAA would draw three copies of the character shown in this bitmask.

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 previous example, the width of the A is 12 bits; that is, a maximum of 12 bits would be written horizontally for this character. The height is also 12 bits.

Normally, a character's origin is at the lower-left corner of the bitmask, and this is the case for the previous example. The origin for a character is what is put at the current character position. For a character with a descender, such as g, (see ″illustration″ on page 48), you need extra bits that lie below the current character position. Therefore, the origin should not be at the lower-left corner. Two values, *xoffset* and *yoffset*, tell how far the character's origin must be moved to bring it to the lower-left corner. For characters with descenders, the *yoffset* value is typically negative.



Using the defraster Subroutine to Describe a Character

Finally, another number for each character indicates how far to the right the current character position must be advanced after drawing the character. This number is usually different from the width and is

labeled the *x* increment. In the previous example of the bitmap for the letter A, the character position is advanced by about 14 pixels to leave a little space between characters.

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 the font contained the A example previously described as its first character, and a bitmask for the letter B as its second, the offset for the letter B would be 12 short integers (the length of the bitmask definition of the letter A). The length and width together determine the number of short integers in a character's definition.

The **font** subroutine selects the font for use when drawing a character string and this font remains the current font until changed by another call to the **font** subroutine.

Default Font (font 0)

The default font is that font associated with font index 0. The user cannot use the **defrasterfont** subroutine to redefine this font. The user can, however, change this font by setting the **$GLFONT0** environment variable.

The default font for the version 3.2 of the operating system supports ISO8859-1 encoding. This font is a fixed-width font, *X* pixels high, *Y* pixels wide, and is a euro-sans-serif type font.

defrasterfont Subroutine

The **defrasterfont** subroutine defines bitmaps for a raster font. The *chars* parameter contains a description of each character in the font. The figure ″Using the ″ on page 48**defrasterfont** Subroutine to Describe a Character 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.
- The amount to add to the current character *x* position after drawing the character.

The *chars* parameter is an array of structures of type `Fontchar`, defined in the standard **/usr/include/gl/gl.h** file. The syntax is as follows:

```
void defrasterfont(Int32 index, Int16 height,
                   Int16 numchars, Fontchar chars[],
                   Int16 numraster, Int16 raster[])
```

The *raster* parameter is an array of bitmap-information shorts given in the *numraster* parameter. It is a one-dimensional array of bitmask bytes ordered from left to right, then bottom to top. Mask bits are left justified in the character's bounding box.

The following code fragment draws the g character (as illustrated in the following ″figure″ on page 48):

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

chars ['g'] = {  724,       8,  9,   0,     -2       9  }
              byte offset   w   h  xoffset yoffset  xinc
              into rasterarray

Int16        []  = {...
rasterarray          ...

position 724 >       0x7E00, 0xC300, 0x0300, 0x0300,
                     0x7F00, 0xC300, 0xC300, 0xC300,
                     0x7E00,
                    ...
                   }
```

font Subroutine
The **font** subroutine selects the font the system uses whenever the **charstr** subroutine draws a text string. The *fontnum* parameter is an index into the font table built by the **defrasterfont** subroutine or loaded with the **loadXfont** subroutine. This font remains the current font until you use the **font** subroutine to select another font. The syntax is as follows:

```
void font(Int32 fontnum)
```

The **font3.c** example program defines a font with three characters: a lowercase j, an arrow, and the Greek letter sigma. The j is assigned to the ASCII value of j, and the arrow and sigma are assigned to ASCII values 1 and 2 (written \001 and \002 in the C code). Two sample strings are then written out, the first of which contains only characters that are defined, while the second contains undefined characters.

Note: When characters are not defined, no error occurs but nothing is printed out for them.

loadXfont Subroutine
The **loadXfont** subroutine loads an Enhanced X-Windows font so that it can be used by GL applications for rendering text. The application must provide a valid font name. There are several ways to obtain a valid font name. The easiest way is to use Enhanced X-Windows subroutines that return lists of installed fonts, with the assumption that a font file is already installed on the system. The **xfonts.c** example program (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) shows subroutines that can be used.

Given a font name, this subroutine searches the file system (along the font path) for that font. If the font name is found, the subroutine loads it into the GL table of defined fonts. In other words, the **loadXfont** subroutine works much like the **defrasterfont** subroutine, except that the character bitmaps are obtained from pre-installed files. The directory path that is searched for fonts can be controlled with Enhanced X-Windows routines. The syntax is as follows:

```
void loadXfont(Int32 id_num, Char8 *name)
```

Font Query Subroutines
The following subroutines return information about the current font (what number it is, how high the characters are, and how long a descender any character has).

getfont Subroutine
The **getfont** subroutine returns the index of the current raster font. The syntax is as follows:

```
Int32 getfont()
```

getheight Subroutine
The **getheight** subroutine returns the maximum height of a character in the current raster font, including ascenders (present in such characters as the letters t and h, which ascend above the baseline) and descenders (present in such characters as the letters y and p, which descend below the baseline). It returns the height in pixels. The syntax is as follows:

```
Int32 getheight()
```

getcpos Subroutine
The **getcpos** subroutine gets the current character position, in screen coordinates relative to the lower-left corner of the window, and writes it into the parameters. The syntax is as follows:

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

getdescender Subroutine
The **getdescender** subroutine returns the longest descender in the current font. It returns the number of pixels that the longest descender goes below the baseline. The syntax is as follows:

```
Int32 getdescender()
```

getfontencoding
The **getfontencoding** subroutine returns the font encoding of the current raster font. The syntax is as follows:

```
void getfontencoding (char * end)
```

getfonttype
The **getfonttype** subroutine returns the font type of the current raster font. Fonts may be either single-byte character set (SBCS) or double-byte character set (DBCS) fonts. The syntax is as follows:

```
int getfonttype()
```

Smoothing Jagged Lines with Antialiasing

This discussion of antialiasing includes the following topics:

- Antialiasing Introduction
- Pixel Coverage
- Improving Intersections
- Depth-Cueing

Other topics that affect or are affected by antialiasing subroutines include Performing Depth-Cueing, Removing Hidden Surfaces, and Working in Color Map and RGB Modes.

List of GL Antialiasing Subroutines

linesmooth	Specifies antialiasing of lines.
pntsmooth	Specifies antialiasing of points.
subpixel	Controls placement of point, line, and polygon vertices.

Antialiasing Introduction

Antialiasing makes lines and points drawn on the display screen appear smooth. You can draw smooth lines using antialiasing with the **linesmooth** subroutine.

When lines are drawn on a raster computer screen, they are often displayed as jagged, especially if they are nearly (but not quite) horizontal or vertical. The reason is that when a line is drawn on the screen, the true mathematical line is approximated by a series of points that happen to lie on the pixel grid. Except for a few special cases (horizontal, vertical, and 45-degree lines), many of the approximating pixels are not on the mathematical line connecting the two pixels.

As an example, consider the worst case: a line that connects (0, 0) to (1279, 1). It moves up one pixel for every 1280 pixels it moves sideways. The line is rendered as two horizontal segments 640 pixels long, one having *y* coordinate 0 and one having *y* coordinate 1. Although the pixels are small, you can easily detect the jump from *y*=0 to *y*=1.

The following **jagged.c** example program illustrates the problem of lines that are displayed as jagged on the computer screen:

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

Int32 vert1[2] = {100, 100};
Int32 vert2[2] = {500, 0};

main()
{
   Int32 xorg, yorg;
```

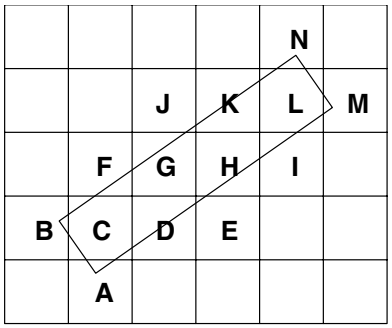
```
    winopen("Jagged");
    doublebuffer();
    gconfig();
    getorigin(&xorg, &yorg);
    while (TRUE) {
        color(BLACK);
        clear();
        color(WHITE);
        vert2[0] = getvaluator(MOUSEX) - xorg;
        vert2[1] = getvaluator(MOUSEY) - yorg;
        bgnline();
        v2i(vert1);
        v2i(vert2);
        endline();
        swapbuffers();
    }
}
```

This example draws a line from the point (100, 100) to the current cursor position. Move the cursor around and notice how jagged the lines are displayed especially when they are nearly vertical or horizontal. Even at angles far from vertical or horizontal, there is some jaggedness, but it is not as noticeable. The jagged effect that you see is called aliasing, and techniques to eliminate or reduce it are called antialiasing.

> Note: Only solid, single, pixel-wide lines can be antialiased, and they cannot be used in conjunction with shading processor functions such as z-buffer, light, shade, and depthcue.

Pixel Coverage

One way to smooth a line is to vary the coloring of the pixels along the path according to how much of each pixel is covered by the line and how much is background color. This is illustrated in the following ″figure″ on page 52. This illustration shows a short line drawn between the pixels (2,2) and (5,4). Each square represents a pixel, and the ideal line segment is the tilted rectangle. This rectangle is exactly one pixel wide, and the centers of pixels C and L are one-half pixel from the end of the rectangle. This is exactly the shape of rectangle that would be drawn parallel to the axes.



A .040510
B .040510
C .878469
D .434259
E .007639
F .141435
G .759952
H .759952
I .141435
J .007639
K .434258
L .878469
M .040510
N .040510

Antialiased Line

Each pixel can be set to only a single color. In the figure ″Antialiased Line″ on page 52, the ideal line hits parts of 14 pixels, and none of them are covered completely. The affected pixels are labeled A through N, and the chart at the side shows the percentage of each pixel that is covered by the ideal rectangle. Here, 87% of C and L are covered, while less than 1 % of E and J are covered.

Other pixels have intermediate-sized intersections. If pure white were color 1.0 and pure black were 0.0, a reasonable antialiased line could be drawn by coloring A and B as `.04051` (almost black), C as `.87847` (almost white), and so on.

linesmooth Subroutine

GL automatically draws antialiased lines in both RGB and color map modes. The **linesmooth** subroutine turns this capability on and off. The syntax is as follows:

```
void linesmooth(Int32 mode)
```

RGB Mode: If antialiasing is turned on in RGB mode, the drawing of antialiased lines proceeds automatically. The drawing hardware automatically performs what is called a “read-modify-write” operation into the frame buffer. That is, it reads the color of a pixel in the frame buffer, computes a new color for that pixel, and then writes it back into the frame buffer. The color that is computed is a blend of the pixel color and the current drawing color, the blend being based on the overlap of the line with the pixel.

To draw antialiased lines in RGB mode, you need only to call the **linesmooth** subroutine, and then proceed to draw as normal.

> Note: In order for antialiased lines and points to appear visually smooth, gamma correction *must* be performed. Gamma correction is performed by loading a gamma-corrected color ramp with the **gammaramp** subroutine. Gamma-corrected ramps are nonlinear ramps from dark colors to light, usually specified with a power function, but sometimes logarithmically. Gamma correction is required to take into account nonlinearities in the display electronics, in the phosphorescence of the display phosphors, in the human visual system, and in the pixel coverage sampling algorithm. If gamma correction is not performed, lines do not appear smooth, but exhibit a *roping* or *braiding* effect, as if the line were composed of separate, intertwining strands.

The **gamma.c** example program demonstrates how a gamma-corrected ramp may be constructed and loaded on the system. A gamma correction factor in the range of 2.4 to 2.7 is suggested.

Color Map Mode: Drawing antialiased lines in color map mode requires greater involvement from the application. In particular, a special color map, called a color ramp, must be loaded. To understand the color ramp, it helps to know what the hardware does in color map mode.

When antialiasing is turned on, the hardware replaces the low-order 4 bits of the current color index with a number ranging from 0 to 15 that represents the fractional pixel coverage. When a line is drawn, the index values stored into the frame buffer are these modified color values. As a result of this replacement of values, the images do not look correct on the screen. To alleviate this problem, the low-order 4 bits of the color map must contain a ramp from the foreground color (the color of the line) and the background color (the color to which the frame buffer was cleared).

In the following example, 16 consecutive cells in the color map are filled with colors that are uniformly spaced between black and white:

```
for (i = 0; i < 16; i++)
{
   mapcolor(144+i, i*17, i*17, i*17);
}
```

This maps color entries 144 through 159 to shades of gray having equal red, green, and blue components of 0, 17, 34, 51, 68, 85, 102, 119, 136, 153, 170, 187, 204, 221, 238, and 255. The starting number, 144, is not completely arbitrary. For antialiasing to work, the starting number must be an exact multiple of 16. The number 144 is large enough that the color map entries affected do not conflict those lower entries in the color map used in many simple applications.

If antialiasing is turned on with the **linesmooth** subroutine, the percentage each pixel is covered by the ideal line is approximated by the hardware, and that number is scaled uniformly into the range of 16 consecutive shades to find the color to paint the pixel. The **smooth.c** example program that follows is a slight modification of the **jagged.c** example program. It behaves similarly except that smooth lines are drawn if the left mouse button is pressed.

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

Int32 vert1[2] = {100, 100};
Int32 vert2[2] = {500, 0};

main()
{
   Int32 xorg, yorg, i;

   winopen("smooth");
   doublebuffer();
   gconfig();
   getorigin(&xorg, &yorg);
   for (i = 0; i < 16; i++)
      mapcolor(144+i, i*17, i*17, i*17);

   while (TRUE) {
      color(BLACK);
      clear();
      if (getbutton(LEFTMOUSE)) {
         linesmooth(TRUE);
         color(144);

      } else {
         linesmooth(FALSE);
         color(WHITE);
      }
      vert2[0] = getvaluator(MOUSEX) - xorg;
      vert2[1] = getvaluator(MOUSEY) - yorg;

      bgnline();
         v2i(vert1);
         v2i(vert2);
      endline();
      swapbuffers();
   }
}
```

Before anything is drawn, the color map entries between 144 and 159 are loaded with shades of gray. Then, in the main loop, the left mouse button is examined, and the color is set either to white, if the **linesmooth** subroutine is turned off, or to 144 if it is turned on.

You can draw multicolor antialiased lines on arbitrary multicolored backgrounds if you load suitable ramps. The **alias.c** example program partitions eight adapter bitplanes into two that store the background image and two that store the line colors, with the remaining four being overridden by the antialiasing hardware. The color map is loaded with ramps from all possible line colors to all possible background colors. The **alias_back.c** example program illustrates the drawing of antialiased lines of multiple foreground colors on a monochrome background. The **alias_fore.c** example program, in contrast, illustrates drawing antialiased lines of a single foreground color on an arbitrary multicolored background.

The accurate selection of ramp intensities is imperative for effective visual appearance. The ramp must include corrections for nonlinearities in the electronics, the screen phosphors, and the human eye. Such corrected ramps are usually referred to as gamma ramps. In the **alias.c** example program, the colors are gamma corrected. The gamma ramp hardware, however, is not used; rather, the corrections are folded in with the color ramps.

Notes:

1. The Supergraphics Processor Subsystem does not support antialiasing in color map mode.
2. The High-Performance 3D Color Graphics Processor does not support antialiasing in RGB mode.

Other topics that affect or are affected by antialiasing subroutines include Performing Depth-Cueing, Removing Hidden Surfaces, and Working in Color Map and RGB Modes.

Improving Intersections

When antialiased lines cross each other, there may be some undesirable presentations because the hardware is interpolating between the line color and the background color. It does not take into account any other lines that may already be drawn. Thus, if a program draws a series of smooth lines all crossing each other at the same point, you might expect the intersection to be almost entirely the color of the line. However, the hardware continues to average the original background color into each new line. The following example program, **linesmooth1.c**, which draws a multirayed star, illustrates the problem:

```
#include <gl/gl.h>
#include <math.h>
#define PI 3.14159265
#define RADIUS 2.0

main()
{
   long i;
   float x, y, radangle;

   keepaspect(1, 1);
   winopen("linesmooth1");
   ortho2(-1.0, 1.0, -1.0, 1.0);
   for (i = 0; i < 16; i++)
      mapcolor(256+i, 17*i, 17*i, 17*i);
   color(BLACK);
   clear();
   color(256);
   linesmooth(TRUE);

   for (i = 0; i <= 180; i += 10) {
      radangle = i*PI/180.0;
      x = RADIUS*cos(radangle);
      y = RADIUS*sin(radangle);
      move2(-x, -y);
      draw2(x, y);
   }
   sleep(20);
}
```

The first ray drawn contains pixels blended with the background color to smooth the line. Each successive line's color is also blended with the original background color and is unaffected, even close to the intersection, by the color resulting from the previous line's blending. The pixels surrounding the center of the star are no brighter than any other line portion that has been averaged with the background color. Antialiased lines in color map mode have no effect on the color of successively drawn lines.

The interpolation problem is also apparent when two lines that are nearly parallel cross each other. One solution to this sort of problem can be obtained by using the z-buffering hardware for color comparisons. Instead of using the z-buffer to draw objects that are nearest the viewer, the z-buffering compares each new pixel color with the existing color for that pixel and causes the brightest color to be drawn. This is not a perfect solution, but it gives good results. The point at the intersection of two lines is at least as bright as the lines were originally.

Example Program With and Without Color Comparison

The following example program, **linesmooth2.c**, draws a pair of intersecting lines both with and without the color comparison. As it runs, press the left mouse button to see the effects of a z-buffer type color comparison.

```
#include <gl/gl.h>
#include <math.h>
#include <device.h>
#define PI 3.14159265
#define RADIUS 300.0
```

```
main()
{
   long i;
   float x, y, radangle;

   keepaspect(1, 1);
   winopen("linesmooth2");
   doublebuffer();
   ortho2(-1.0, 1.0, -1.0, 1.0);
   RGBmode();
   gconfig();
   frontbuffer(TRUE);
   wmpack(0xffffffff);
   cpack(0);
   clear();
   frontbuffer(FALSE);
   cmode();
   gconfig();

   for (i = 0; i < 16; i++)
      mapcolor(48+i, 17*i, 17*i, 17*i);
   linesmooth(TRUE);
   for (i = 0; i < 1000000; i++) {
      if (getbutton(LEFTMOUSE)) {
      zbuffer(TRUE);
      zsource(ZSRC_COLOR);
      zfunction(ZF_GEQUAL);
      } else {
      zbuffer(FALSE);
      }
      color(BLACK);
      clear();
      color(48);

      move2(-1.0, 0.0);
      draw2(1.0, 0.0);
      move2(-1.0, -sin(i/30.0)*.05);
      draw2(1.0, sin(i/30.0*.05);
      swapbuffers();
   }
}
```

The **linesmooth2.c** example program has several interesting features. The hardware that does the comparison to find the largest color index actually compares the entire 24-bit contents of the pixel. The same physical memory is used for RGB information as is used for color indexes, but the color index data occupies only the low-order 12 bits. If there is garbage in the high-order bits, perhaps left behind by previous windows, this has no effect on the displayed values. Because the comparisons are made on the full 24 bits, the garbage contents can have an effect on the results of the comparisons.

For this reason, it is a good idea to clear out all 32 bits completely before starting. This is done by putting the system in RGB mode, and then writing a zero (for all 32 bits) with a writemask of 0xffffffff (all 32 bits enabled). This guarantees that everything in the high-order bits of the frame buffer is set to zero. Because the **linesmooth2.c** example program is double buffered, this must be done for both the front and back buffers by setting `frontbuffer(TRUE)`.

Enabling Color Comparison

Three things must be done to enable the z-buffer color comparison properly:

- The z-buffer must be turned on by setting `zbuffer(TRUE)`.
- The z-buffer must be set to do a color comparison instead of a depth comparison with `zsource(ZSRC_COLOR)`.
- The color comparison must be changed by setting `zfunction(ZF_GEQUAL)`.

The comparison function is ZF_GEQUAL (it could also be ZF_GREATER), so that the new value is written into the pixel if its value is greater than or equal to the current value. (In standard z-buffer comparisons, values are written if they are closer to the eye: ZF_LESS or ZF_LEQUAL.)

pntsmooth Subroutine

The **pntsmooth** subroutine draws antialiased points. The syntax is as follows:

```
void pntsmooth(Int32 mode)
```

subpixel Subroutine

The **subpixel** subroutine controls the placement of point, line, and polygon vertices in screen coordinates. The default value of the *bool* parameter is False, causing vertices to be snapped to the center of the nearest pixel after they have been transformed to screen coordinates. The **subpixel** subroutine is typically set to True while smooth points or smooth lines are being drawn.

The syntax is as follows:

```
void subpixel(Int32 bool)
```

Depth-Cueing

It is possible to draw lines that are both antialiased and depth-cued in color map mode, but not in RGB mode The depth-cueing hardware maps the transformed *z* component linearly into a region of the color map. If the color map is arranged as a series of 16-entry ramps, each beginning at a multiple of 16, and each mapping a range from the background color to a series of brighter and brighter colors, depth-cueing and antialiasing work together.

First, the depth-cueing calculation gives a position within the map corresponding to how bright a fully illuminated pixel on a line should be. Then the antialiasing hardware calculates a percentage pixel coverage, and the appropriate entry from the 16 color range is chosen. Although it is an approximation, this method gives reasonably good results.

Drawing Wire Frame Curves and Surface Patches

This section discusses the following topics:

- Wire Frame Curves and Surface Patches Introduction
- Curve Mathematics including Bezier Cubic Curve, Cardinal Spline Cubic Curve, and B-Spline Cubic Curve
- Drawing Curves
- Drawing Surfaces

List of GL Wire Frame Curve and Surface Patch Subroutines

crv	Draws a cubic spline curve.
crvn	Draws a series of cubic spline curves.
curvebasis	Sets the current cubic spline curve basis matrix.
curveit	Draws a curve segment by iterating the forward difference matrix.
curveprecision	Sets the number of line segments that compose a cubic spline curve.
defbasis	Defines a cubic spline basis matrix.
patch	Draws a cubic spline surface patch.
patchbasis	Sets the current spline surface basis matrices.
patchcurves	Sets the number of curves used to represent a patch.
patchprecision	Sets the precision at which curves are drawn.
rcrv	Draws a rational cubic spline curve.

rcrvn	Draws a series of rational curve segments.
rpatch	Draws a rational cubic spline surface patch.

Wire Frame Curves and Surface Patches Introduction

This section describes the models, mathematics, and programming statements used for drawing curves and surfaces that were available before the NURBS functions that have been introduced in the latest release of GL. These techniques and GL functions are still supported for compatibility with programs written for earlier versions of GL.

You draw a curve segment by specifying:

- A set of four control points.
- A basis, which defines how the system uses the control points to determine the shape of the segment.

You create complex curved lines by joining several curve segments end to end. The curve facility provides the means for making smooth joints between the segments.

Three-dimensional surfaces, or patches, are represented by a wire frame of curve segments. You draw a patch by specifying:

- A set of 16 control points.
- The number of curve segments to be drawn in each direction of the patch.
- The two bases that define how the control points determine the shape of the patch.

You can create complex surfaces by joining several patches into one large patch.

Curve Mathematics

The mathematical basis for the GL curve facility is the parametric cubic curve. The curves in most applications are too complex to be represented by a single curve segment and instead must be represented by a series of curve segments joined end to end. To create smooth joints, you must control the positions and curvatures at the endpoints of curve segments. Parametric cubic curves are the lowest order representation of curve segments that provide continuity of position, slope, and curvature at the point where two curve segments meet.

In the following equation, a parametric cubic curve has the property that *x*, *y*, and *z* can be defined as third-order polynomials for variable *t*:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$$

$$z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$$

A cubic curve segment is defined over a range of values for *t* (usually 0 <= *t* <= 1), and can be expressed as a vector product as in this equation:

$$C(t) = at^3 + bt^2 + ct + d$$

$$= \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

$$= TM$$

GL approximates the shape of a curve segment with a series of straight line segments. The endpoints for all the line segments can be computed by evaluating the vector product C(t) for a series of t values between 0 and 1. The shape of the curve segment is determined by the coefficients of the vector product, which are stored in column vector M. These coefficients can be expressed as a function of a set of four control points. Thus, the vector product becomes

```
C(t) = T M = T (B G)
```

where G is a set of four control points, or the geometry, and B is a matrix called the basis. The basis matrix is determined from a set of constraints that express how the shape of the curve segment relates to the control points. For example, a constraint might be that one endpoint of the curve segment is located at the first control point; or the tangent vector at that endpoint lies on the line segment formed by the first two control points. When the vector product C is solved for a particular set of constraints, the coefficients of the vector product are identified as a function of four variables (the control points). Then, given four control points, you can use the vector product to generate the points on the curve segment.

There are three classes of cubic curves: Bezier, Cardinal spline, and B-spline. Each has a set of constraints that define its class, plus a basis matrix derived from those constraints that you can use to draw curve segments.

Bezier Cubic Curve

A Bezier cubic curve segment passes through the first and fourth control points and uses the second and third points to determine the shape of the curve segment. Of the three kinds of curves, the Bezier form provides the most intuitive control over the shape of the curve. The Bezier basis matrix is derived from the following four constraints:

One endpoint of the segment is located at p(1):

$$Bezier(0) = p_1$$

The other endpoint is located at p(4):

$$Bezier(1) = p_4$$

The first derivative, or slope, of the segment at one endpoint is equal to this value:

$$Bezier(0)' = 3(p_2 - p_1)$$

The first derivative at the other endpoint is equal to this value:

$$Bezier(1)' = 3(p_4 - p_3)$$

Solving for these constraints yields this equation:

$$Bezier(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

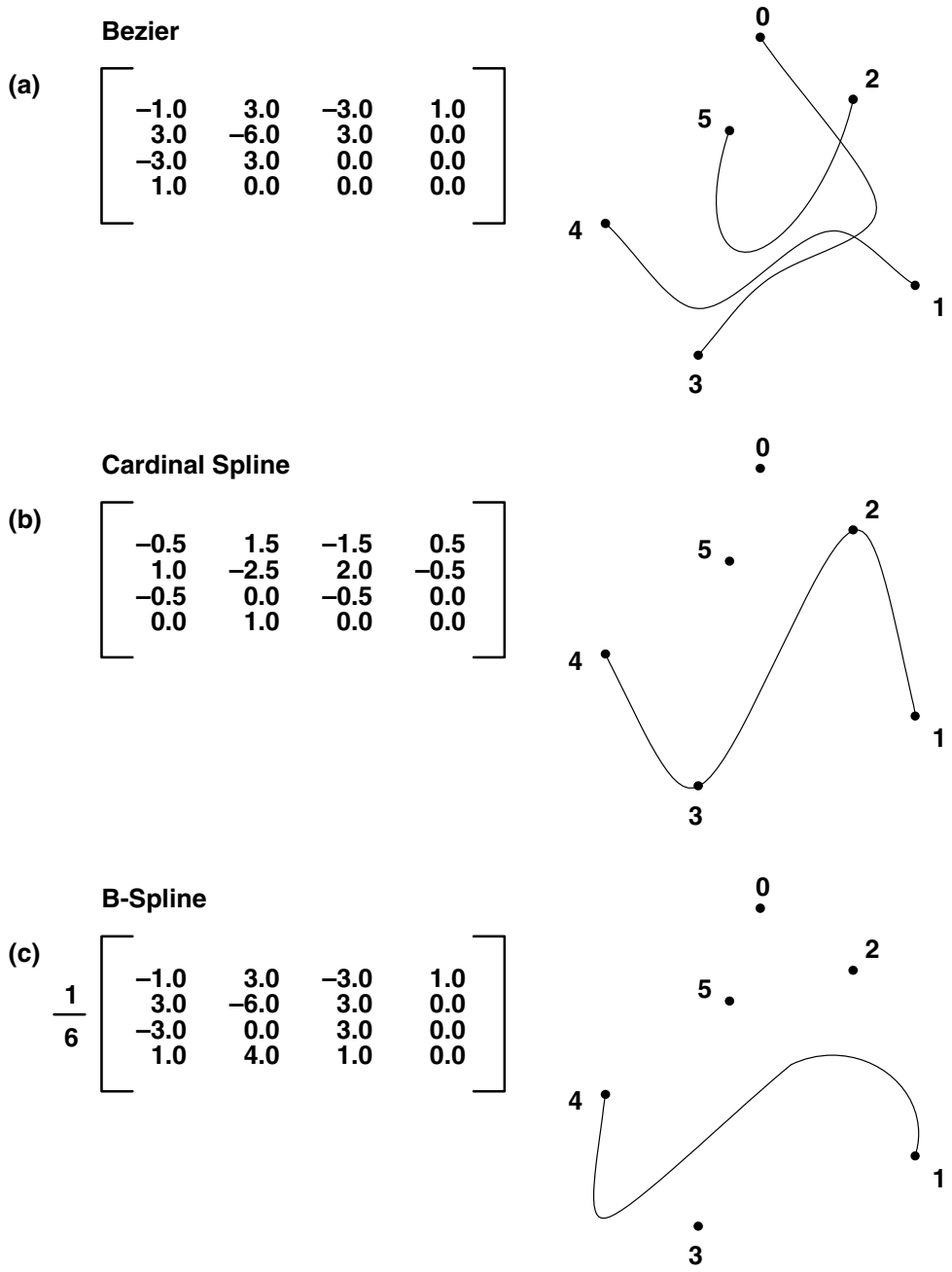
$$= TM_bG_b$$

You can generate all the points on the Bezier cubic curve segment from p(1) to p(4) by evaluating Bezier(t) for 0 <= t <= 1. It is more efficient, however, to construct a forward difference matrix that generates the points in a curve segment incrementally.

The following figure, Bezier, Cardinal, and B-Spline Curves, shows three Bezier curve segments. The first segment uses points 0, 1, 2, and 3 as control points. The second uses 1, 2, 3, and 4. The third uses 2, 3, 4, and 5. You can use the technique of overlapping sets of control points more effectively with the following two classes of cubic curves to create a single large curve from a series of curve segments.

Cardinal Spline Cubic Curve

In the following figure, a spline curve segment passes through the two interior control points and is continuous in the first derivative at the points where segments meet. The curve segment starts at p(2) and ends at p (3), and uses p(1) and p(4) to define the shape of the curve.



Bezier, Cardinal, and B-Spline Curves

Three different curves are shown with appropriate basis matrices. With the Bezier basis matrix, three sets of overlapping control points result in three separate curve segments. With the Cardinal spline and B-spline matrices, the same overlapping sets of control points result in three joined curve segments.

The Cardinal spline basis matrix is derived from the following four constraints:

$$\text{Cardinal}(0) = p_2$$

$$\text{Cardinal}(1) = p_3$$

$$\text{Cardinal}(0)' = a(p_3 - p_1)$$

$$\text{Cardinal}(1)' = a(p_4 - p_2)$$

The scalar coefficient a must be positive; it determines the length of the tangent vector at point:

$$p2 : \text{tangent2} = a(p_3 - p_1)$$

and at >point:

$$p3 : \text{tangent3} = a(p_4 - p_2)$$

Solving for these constraints yields the following equation:

$$\text{Cardinal}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -a & 2-a & -2+a & a \\ 2a & -3+a & 3-2a & -a \\ -a & 0 & a & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_bG_b$$

The three joined Cardinal spline curve segments in the Bezier, Cardinal, and B-Spline Curves figure use the same three sets of control points as the Bezier curve segments. Many different bases have Cardinal spline properties. You can derive the different bases by trying different values of *a*.

B-Spline Cubic Curve

In general, a B-spline curve segment does not pass through any control points, but is continuous in both the first and second derivatives at the points where segments meet. Thus, a series of joined B-spline curve segments is smoother than a series of Cardinal spline segments (see the Bezier, Cardinal, and B-Spline Curves figure).

The B-spline basis matrix is derived from the following four constraints:

$$\text{B-spline}(0)' = \frac{(p_3 - p_1)}{2}$$

$$\text{B-spline}(1)' = \frac{(p_4 - p_2)}{2}$$

$$\text{B-spline}(0)'' = p_1 - 2p_2 + p_3$$

$$\text{B-spline}(1)'' = p_2 - 2p_3 + p_4$$

Solving for these constraints yields the following equation:

$$\text{B-spline}(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

$$= TM_BG_B$$

Drawing Curves

Drawing a curve segment on the screen involves four steps:

1. Define and name a basis matrix with the **defbasis** subroutine.
2. Select a defined basis matrix as the current basis matrix with the **curvebasis** subroutine.
3. Specify the number of line segments used to approximate each curve segment with the **curveprecision** subroutine.
4. Draw the curve segment using the current basis matrix, the current curve precision, and the four control points with the **crv** subroutine. The **rcrv** subroutine draws a rational curve.

defbasis Subroutine

The **defbasis** subroutine defines and names a basis matrix to generate curves and patches. The value of the *mat* parameter is saved and is associated with the *id* parameter. Use the *id* parameter in subsequent calls to the **curvebasis** and **patchbasis** subroutines. The syntax is as follows:

```
void defbasis(Int32 id, Matrix mat)
```

curvebasis Subroutine

The **curvebasis** subroutine selects a basis matrix (defined by the **defbasis** subroutine) as the current basis matrix to draw curve segments. The syntax is as follows:

```
void curvebasis(Int32 basis_id)
```

curveprecision Subroutine

The **curveprecision** subroutine specifies the number of line segments used to draw a curve. Whenever the **crv**, **crvn**, **rcrv**, or **rcrvn** subroutine executes, a number of straight line segments (the value of the *nsegments* parameter) approximates each curve segment. The greater the value of the *nsegments* parameter, the smoother the curve, but the longer the drawing time. The syntax is as follows:

```
void curveprecision(Int16 nsegments)
```

crv Subroutine

The **crv** subroutine draws the curve segment using the current basis matrix, the current curve precision, and the four control points specified in the *points* parameter. The syntax is as follows:

```
void crv(Coord points[4][3])
```

When you issue the **crv** command, a matrix is built from the geometry, the current basis, and the current precision:

$$M = F_{precision} M_{basis} G_{geom}$$

$$= \begin{bmatrix} \frac{6}{n^3} & 0 & 0 & 0 \\ \frac{6}{n^3} & \frac{2}{n^2} & 0 & 0 \\ \frac{1}{n^3} & \frac{1}{n^2} & \frac{1}{n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} M_{basis} G_{geom}$$

where n = the current precision. The bottom row of the resulting transformation matrix identifies the first of n points that describe the curve. To generate the remaining points in the curve, the following algorithm is used to iterate the matrix as a forward difference matrix. The third row is added to the fourth row, the second row is added to the third row, and the first row is added to the second row. The fourth row is then output as one of the points on the curve.

```
/* This is the forward difference algorithm */
/* M is the current transformation matrix */
move (M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
/* iteration loop */
for (cnt = 0; cnt < iterationcount; cnt++) {
   for (i=3; i>0; i--)
      for (j=0; j<4; j++)
         M[i][j] = M[i][j] + M[i-1][j];
   draw(M[3][0]/M[3][3], M[3][1]/M[3][3], M[3][2]/M[3][3]);
}
```

Each iteration draws one line segment of the curve segment. If the precision matrix on the previous page is iterated as a forward difference matrix, it generates the sequence of points:

$$(0, 0, 0, 1); ((\frac{1}{n})^3, (\frac{1}{n})^2, \frac{1}{n}, 1); ((\frac{2}{n})^3, (\frac{2}{n})^2, \frac{2}{n}, 1); ((\frac{3}{n})^3, (\frac{3}{n})^2, \frac{3}{n}, 1); ...$$

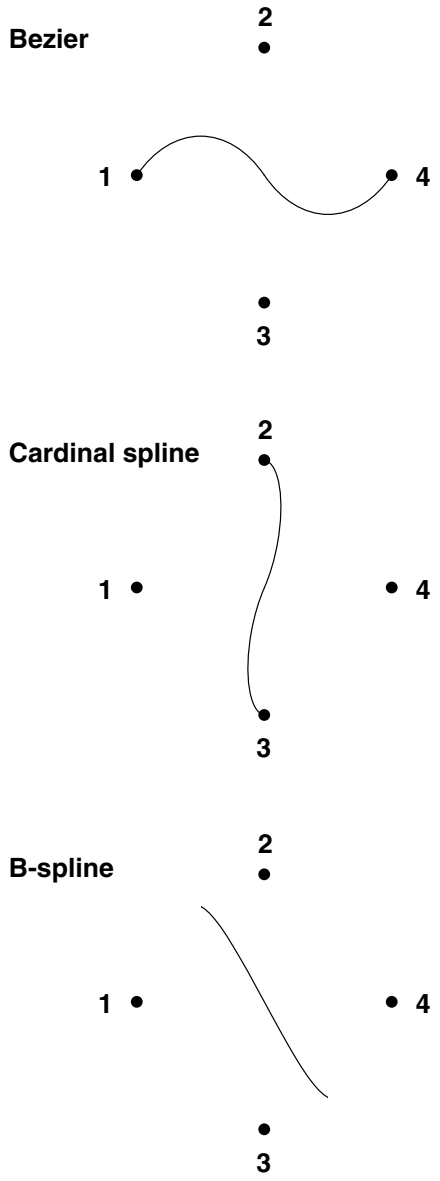
This is the same sequence of points generated by the equation:

$$t = 0, \frac{1}{n}, \frac{2}{n}, \frac{3}{n}, ...$$

for the vector

$$(t^3, t^2, t, 1)$$

The example program **curve2.c** (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) draws the three curve segments in the figure entitled Curve Segments. All use the same set of four control points, which is contained in the *geom1* parameter. The three basis matrix arrays (`beziermatrix`, `cardinalmatrix`, and `bsplinematrix`) contain the values outlined in the Bezier, Cardinal, and B-Spline Curves figure.



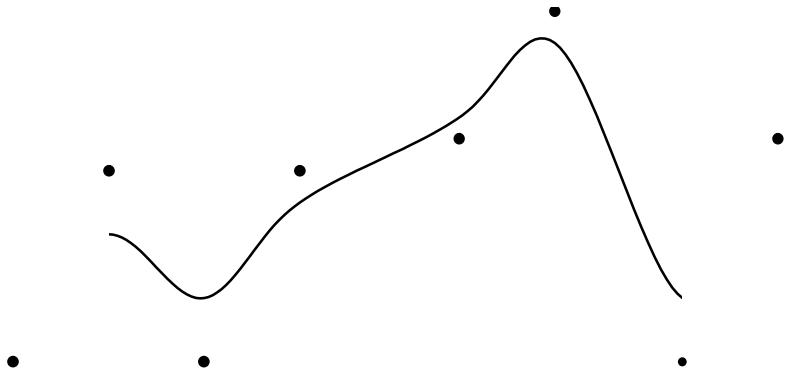
Curve Segments

Before the **crv** or **rcrv** subroutine is called, a basis and precision matrix must be defined. This is also true if the routines are compiled into an object.

Each of the curve segments in the previous figure uses the same set of four control points and the same precision, but a different basis matrix.

crvn Subroutine

The **crvn** subroutine takes a series of control points and draws a series of cubic spline or rational cubic spline curve segments using the current basis and precision; the **rcrvn** subroutine draws rational splines. The control points specified in the *geom* parameter determine the shapes of the curve segments and are used four at a time. If the current basis is a B-spline, Cardinal spline, or basis with similar properties, the curve segments are joined end to end and appear as a single curve. Calling the **crvn** subroutine has the same effect as calling the **crv** subroutine with overlapping control points (see the Uniform Cubic B-Spline With No Rational Component figure). The syntax is as follows:

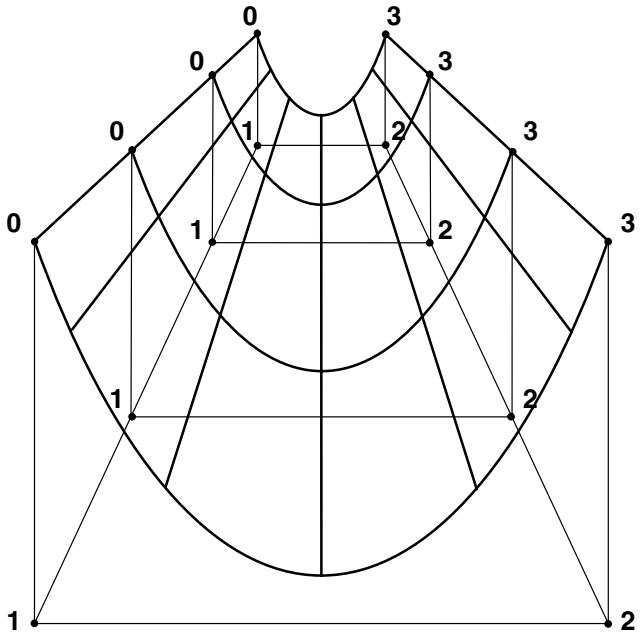


Uniform Cubic B-Spline with No Rational Component

```
void crvn(Int32 n, Coord geom[][3])
```

When you issue this subroutine with a Cardinal spline or B-spline basis, it produces a single curve. However, a **crvn** subroutine issued with a Bezier basis produces several separate curve segments.

As with the **crv** and **rcrv** subroutines, a precision and basis must be defined before calling the **crvn** or **rcrvn** subroutine. This is true even if the routines are compiled into objects. The example program **curve2.c** (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) draws the three joined curve segments in the Bezier Surface Patch figure using the **crvn** subroutine. The *geom2* parameter contains six control points.



Bezier Surface Patch

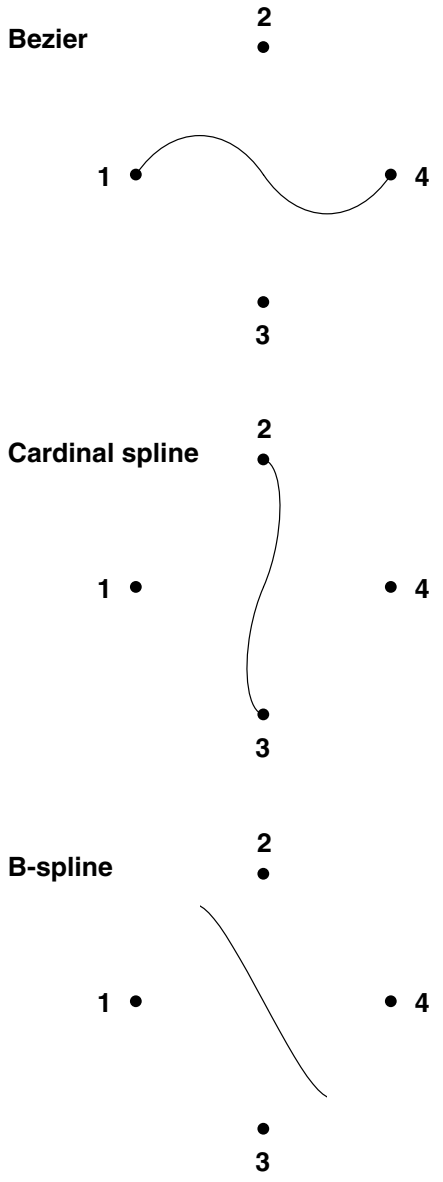
curveit Subroutine

The iteration loop of the forward difference algorithm is implemented in the graphics pipeline. The **curveit** subroutine provides direct access to this facility, making it possible to generate a curve directly from a forward difference matrix. This subroutine iterates the current matrix (the one on top of the matrix stack) as many times as indicated in the *count* parameter. Each iteration draws one of the line segments that approximate the curve. The syntax is as follows:

```
void curveit(Int16 count)
```

The **curveit** subroutine does not execute the initial move in the forward difference algorithm. A `move(0.0,0.0,0.0)` must precede the **curveit** subroutine so that the correct first point is generated from the forward difference matrix.

This example program **curve3.c** (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) draws the Bezier curve segment shown in the figure entitled Curve Segments using the **curveit** subroutine. The Cardinal spline and B-spline curve segments could be drawn using a similar sequence of commands - only the basis matrix would be different.



Curve Segments

Rational Curves

Cubic splines have been the focus of discussion. Cubic splines are splines whose *x*, *y*, and *z* coordinates can be expressed as a cubic polynomial in *t*.

GL actually works in homogeneous coordinates *x*, *y*, *z*, and *w*, where 3-D coordinates are given by *xw*, *yw*, and *zw*. The *w* coordinate is normally the constant 1, so the homogeneous character of the system is hidden.

In fact, the *w* coordinate can also be expressed as a cubic function of *t*, so that the 3-D coordinates of points along the curve are given as a quotient of two cubic polynomials. The only constraint is that the denominator for all three coordinates must be the same. When *w* is not the constant 1, but some cubic polynomial function of *t*, the curves generated are usually called parametric rational cubic curves.

A circle is a useful example. There is no cubic spline that exactly matches any short segment of a circle, but if *x*, *y*, *z*, and *w* are defined in this equation>:

$$x(t) = t^2 - 1$$

$$y(t) = 2t$$

$$z(t) = 0$$

$$w(t) = t^2 + 1$$

the real coordinates, as shown in this equation,

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) = \left(\frac{t^2 - 1}{t^2 + 1}, \frac{2t}{t^2 + 1}, 0\right)$$

all lie on the circle with center at (0,0,0) in the *x-y* plane with radius 1 (exactly). All the conic sections (ellipses, hyperbolas, parabolas) can be similarly defined.

For rational splines, the basis definitions and precision specifications are identical to those for cubic splines. The only difference is that the geometry matrix must be specified in four-dimensional homogeneous coordinates. This is done with the **rcrv** subroutine.

rcrv Subroutine

The **rcrv** subroutine draws a rational curve segment using the current basis matrix, the current curve precision, and the four control points specified in the its parameter. The syntax is as follows:

```
void rcrv(Coord geom[4][4])
```

The **rcrv** subroutine is exactly analogous to the **crv** subroutine, except that *w* coordinates are included in the control point definitions.

rcrvn Subroutine

The **rcrvn** subroutine takes a series of control points given by the value of the *n* parameter and draws a series of parametric rational cubic curve segments, using the current basis and precision. The control points specified in the *geom* parameter determine the shapes of the curve segments and are used four at a time. The syntax is as follows:

```
void rcrvn(Int32 n, Coord geom[][4])
```

Drawing Surfaces

The method for drawing surfaces is similar to that for drawing curves. A surface patch appears on the screen as a wire frame of curve segments. A set of user-defined control points determines the shape of the patch. A complex surface consisting of several joined patches can be created by using overlapping sets of control points and the B-spline and Cardinal spline curve bases shown in the Bezier, Cardinal, and B-Spline Curves figure.

The mathematical basis for the GL surface facility is the parametric bicubic surface.

The parametric equation for *x* is:

$$x(u,v) = a_{11}u^3v^3 + a_{12}u^3v^3 + a_{13}u^3v + a_{14}u^3$$
$$+ a_{21}u^2v^3 + a_{22}u^2v^2 + a_{23}u^2v + a_{24}u^2$$
$$+ a_{31}uv^3 + a_{32}uv^2 + a_{33}uv + a_{34}u$$
$$+ a_{41}v^3 + a_{42}v^2 + a_{43}v + a_{44}$$

(The equations for *y* and *z* are similar.) The points on a bicubic patch are defined by varying the parameters u and v from 0 to 1. If one parameter is held constant and the other varied from 0 to 1, the result is a cubic curve. Thus, a wire frame patch can be created by holding u constant at several values and using the GL curve facility to draw curve segments in one direction, and then doing the same for v in the other direction.

There are five steps involved in drawing a surface patch:

1. The appropriate bases matrices are defined using the **defbasis** subroutine. A Bezier basis provides intuitive control over the shape of the patch. The Cardinal spline and B-spline bases shown in the Bezier, Cardinal, and B-Spline Curves figure allow smooth joints to be created between patches.
2. A basis for each of the directions in the patch, u and v, must be specified with the **patchbasis** subroutine.

   Note: The u basis and the v basis do not have to be the same.
3. The number of curve segments to be drawn in each direction is specified by the **patchcurves** subroutine. A different number of curve segments can be drawn in each direction.
4. The precisions for the curve segments in each direction must be specified with the **patchprecision** subroutine. The precision is the minimum number of line segments approximating each curve segment and can be different for each direction. The actual number of line segments is a multiple of the number of curve segments being drawn in the opposing direction. This guarantees that the u and v curve segments forming the wire frame actually intersect.
5. The surface patch is actually drawn with the **patch** subroutine. The parameters contain the 16 control points that govern the shape of the patch. The value of the *geomx* parameter is a 4x4 matrix containing the *x* coordinates of the 16 control points; the *geomy* parameter contains the *y* coordinates; the *geomz* parameter contains the *z* coordinates. The curve segments in the patch are drawn using the current linestyle, linewidth, color, and writemask.

The **rpatch** subroutine draws a rational surface patch.

patchbasis Subroutine

The **patchbasis** subroutine sets the current basis matrices (defined by the **defbasis** subroutine) for the parametric directions of a surface patch as given in the *uid* and *vid* parameters. The syntax is as follows:

```
void patchbasis(Int32 uid, Int32 vid)
```

patchcurves Subroutine

The **patchcurves** subroutine sets the current number of curves in both directions as given in the *ucurves* and *vcurves* parameters that represent a patch as a wire frame. The syntax is as follows:

```
void patchcurves(Int32 ucurves, Int32 vcurves)
```

patchprecision Subroutine

The **patchprecision** subroutine sets the precision at which curves defining a wire frame patch are drawn. The u and v directions for a patch specify the precisions independently. Patch precisions specify the minimum number of line segments used to draw a patch. The syntax is as follows:

```
void patchprecision(Int32 usegments, Int32 vsegments)
```

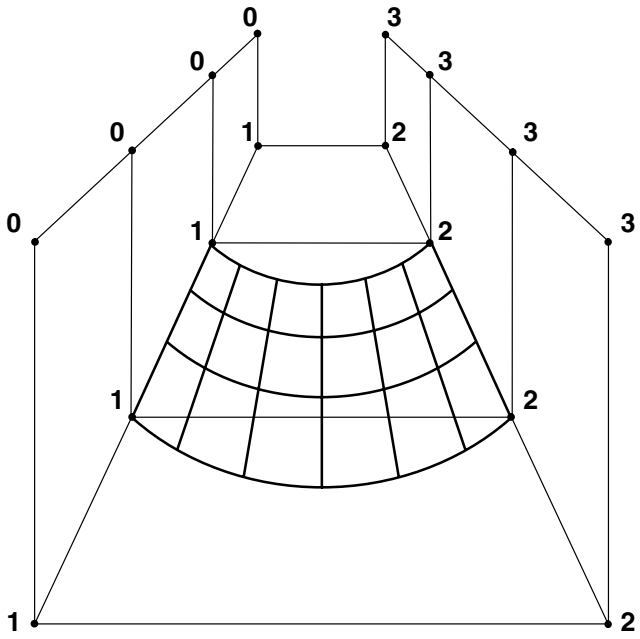
patch and rpatch Subroutines

The **patch** and **rpatch** subroutines draw a surface patch using the current values set by the **patchbasis**, **patchprecision**, and **patchcurves** subroutines. The **rpatch** subroutine draws a rational surface patch. The control points given in the *geomx*, *geomy*, and *geomz* parameters determine the shape of the patch. The control point given in the *geomw* parameter specifies the rational component of the patch to the **rpatch** subroutine. The syntax for the **patch** and **rpatch** subroutines is as follows:

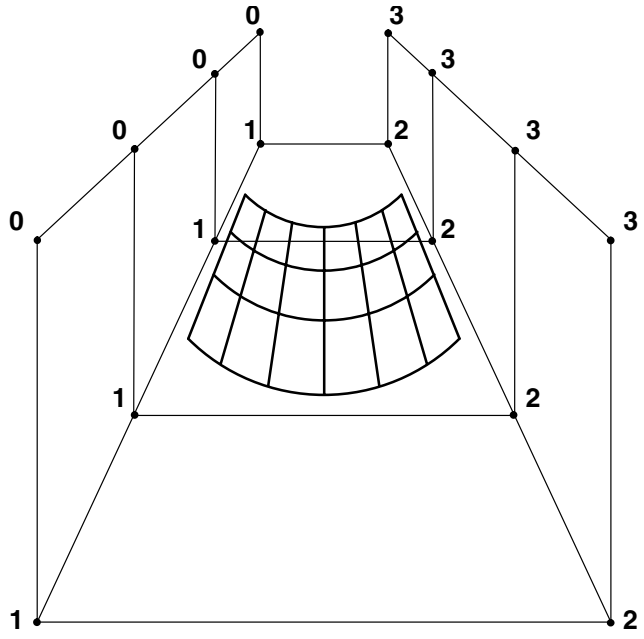
```
void patch(Matrix geomx, Matrix geomy, Matrix geomz)

void rpatch(Matrix geomx, Matrix geomy,
            Matrix geomz, Matrix geomw)
```

The Bezier Surface Patch, Cardinal Spline Surface Patch, and B-Spline Surface Patch figures show the same number of curve segments and the same precisions but different basis matrices. All three use the same set of 16 control points.



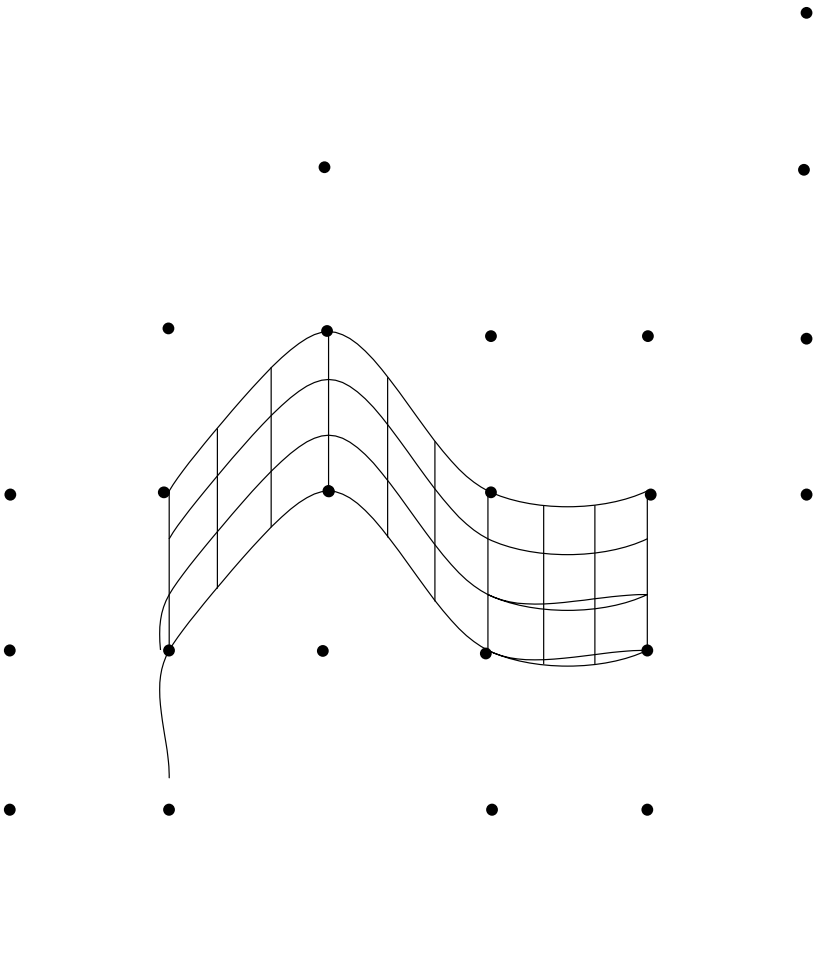
Cardinal Spline Surface Patch



B-Spline Surface Patch

The example program **patch1.c** (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) draws three surface patches similar to those shown in the foregoing figures.

You can join patches to create a more complex surface by using the Cardinal spline or B-spline bases and by overlapping sets of control points. The surface in the Joined Patches figure consists of three joined patches and was drawn using a Cardinal spline basis matrix.



Joined Patches

Drawing NURBS Curves and Surface Patches

The section on NURBS curves and surface patches includes discussions on the following:

- NURBS Curves and Surfaces Introduction
- B-Spline Curves and Surfaces
- NURBS Interface
- NURBS Surface Description
- Trimming
- Controlling Display Properties

List of GL NURBS Curve and Surface Patch Subroutines

bgnsurface	Marks the beginning of a NURBS surface definition.
bgntrim	Marks the beginning of a NURBS surface trimming loop.
endsurface	Marks the end of a NURBS surface definition.
endtrim	Marks the end of a NURBS surface trimming loop.
getnurbsproperty	Returns the current value of a trimmed NURBS surfaces display property.
nurbscurve	Controls the shape of a NURBS trimming curve.

nurbssurface	Controls the shape of an untrimmed NURBS surface.
pwlcurve	Describes a piecewise linear trimming curve for NURBS surfaces.
setnurbsproperty	Sets the property for display of trimmed NURBS surfaces.

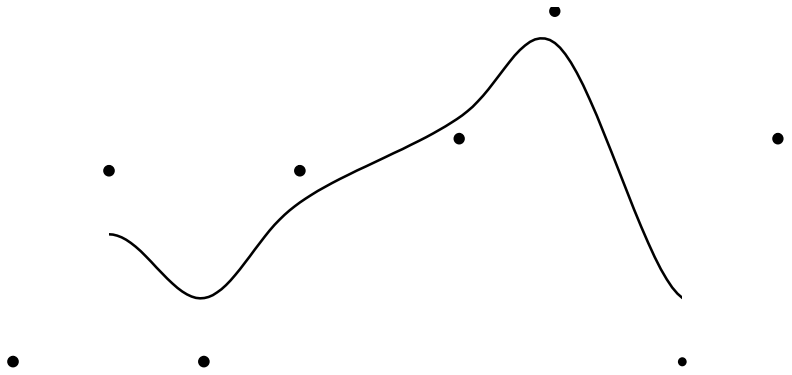
NURBS Curves and Surfaces Introduction

GL provides subroutines that draw parametric non-uniform rational B-spline surfaces (NURBS) that can be trimmed with NURBS curves and piecewise linear curves.

As you can with most other graphics library primitives, you can transform NURBS curves and surfaces with the standard GL modeling commands. You must use the standard lighting models when rendering NURBS curves and surfaces.

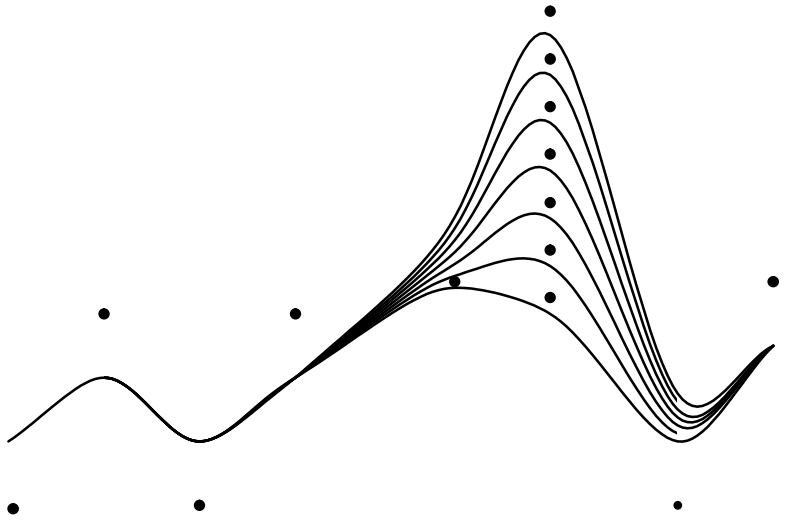
B-Spline Curves and Surfaces

The following figure illustrates a spline with a set of 8 control points. Notice how the spline (or curve) is attracted to the control points, but does not necessarily pass through any of them.



Uniform Cubic B-Spline with No Rational Component

The figure entitled Effects of Moving a Control Point illustrates the result of moving the sixth control point from the left to a series of locations, and the corresponding B-splines created as the one control point moves. Notice that moving the control point affects only a portion of the curve near the control point. This is an important property of B-splines; the influence of the control points is local.

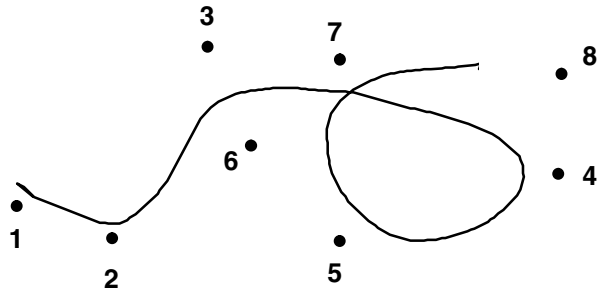


Effects of Moving a Control Point

In fact, for cubic B-splines, each small segment of the curve is controlled by the positions of 4 control points. In this example, the curve is actually drawn as 5 small segments. The first is controlled by points 1, 2, 3, 4; the second by 2, 3, 4, 5; and so on.

The last segment is controlled by control points 5, 6, 7, and 8. When the sixth control point is moved, the only parts of the spline affected are those controlled by points 3, 4, 5, 6, points 4, 5, 6, 7, and points 5, 6, 7, 8.

In the two preceding examples, the control points are evenly spaced in the horizontal direction. This is not necessary, as illustrated In the following figure, Uneven Control Point Spacing.



Uneven Control Point Spacing

Any number of control points greater than 4 can be used to define a cubic B-spline. The spline is actually drawn in segments, each of which is controlled by successive sets of 4 control points.

Trimmed NURBS surfaces are a convenient means of representing curving, bent, and cut surfaces. The bends and curves of the surface are represented by a polynomial mapping of a 2-D space (the s-t plane, or domain) into 3-D space. A rational polynomial mapping can be achieved in 3-space if the s-t plane is mapped into projective 4-space. Cuts and holes in the surface can be achieved by the use of trimming loops, which are closed curves in s-t space. Trimming loops help describe what subset of the s-t plane should actually be mapped into 3-space (drawn on the monitor). Trimming loops themselves may be specified as NURBS curves or as piecewise linear curves.

NURBS Interface

To describe an untrimmed NURBS surface, you must specify these controlling factors:

- A set of nondecreasing knot values in both the s and t directions.
- The order (which is the degree + 1) of the surface in both directions.
- A rectangular set of control points.

The control points can be either three- or four-dimensional, corresponding respectively to polynomial (sometimes called nonrational) and rational surfaces. In three dimensions, the coordinates have the form (*x*, *y*, *z*), and in four, (*wx*, *wy*, *wz*, *w*).

Certain dependencies exist between the surface orders, the knot counts, and the number of control points; that is to say, you specify the surface orders and the knot counts in order to obtain the control points. If Os and Ot are the surface orders in the s and t directions, and if Ks and Kt are the knot counts in those directions, then the control points must form a rectangular array of size (Ks - Os)*(Kt - Ot).

NURBS Surface Description

You define an untrimmed NURBS surface with the **nurbssurface** subroutine as shown in this example program:

```
nurbssurface (
   Int32 sknot_count,   /* of s knots */
   Float64 s_knot[],   /* non-decreasing knot values in s */
   Int32 tknot_count,   /* number of t knots */
   Float64 t_knot[],   /* non-decreasing knot values in t*/
   Int32 s_byte_stride, /* offset to next control point */
            /* in the s direction */
   Int32 t_byte_stride, /* offset to next control point*/
            /* in the t direction */
   double *ctlarray,   /* pointer to first control point */
   Int32 s_order,   /* surface order in s parameter */
   Int32 t_order,   /* surface order in t parameter */
   Int32 type      /* rational or polynomial */
)
```

This implementation of NURBS surfaces supports up to order 4. Trimming curves can be up to order 8.

Many of the parameters in the preceding example are explained in the following:

`s_knot[]`	An array of length `sknot_count`
`t_knot[]`	An array of length `tknot_count`
`s_order`	The order of the surface in the s direction
`t_order`	The order of the surface in the t direction
`type`	One of the constants N_XYZ or N_XYZW (defined in the **gl/gl.h** file) depending on whether the control points are nonrational (3 coordinates), or rational (4 coordinates).

The description of control points is somewhat unusual. The *s_byte_stride* parameter indicates the offset (in bytes) between successive control points in the s direction, and *t_byte_stride* does the same thing for the t direction. This interface is powerful in that the only requirement is that the *x*, *y*, *z*, and possibly *w* coordinates are placed in successive memory locations. The data may be a part of a larger data structure, or the points may form part of a larger array. For example, suppose the data appears as follows:

```
struct ptdata
{
   Int32 tag1, tag2;
   float x, y, z, w;
}   points[5][6];
```

Then the *s_byte_stride* parameter should be set to `sizeof(struct ptdata)`, and the *t_byte_stride* parameter should be set to `6*sizeof(struct ptdata)`, and the *ctlarray* parameter should be `ptdata` or `&(points[0][0].x)`.

As another example, suppose that the data were declared as previously, but only a square of 4 by 4 control points is needed from the middle of the array including everything between and including `points[1][1]` and `points[4][4]`. In that case, the *s_byte_stride* and *t_byte_stride* parameters are as previously, but the *ctlarray* parameter is set to `&(points[1][1].x)`.

> Note: In both examples, the type is N_XYZW because the data includes the homogeneous *w* coordinate.

Trimming

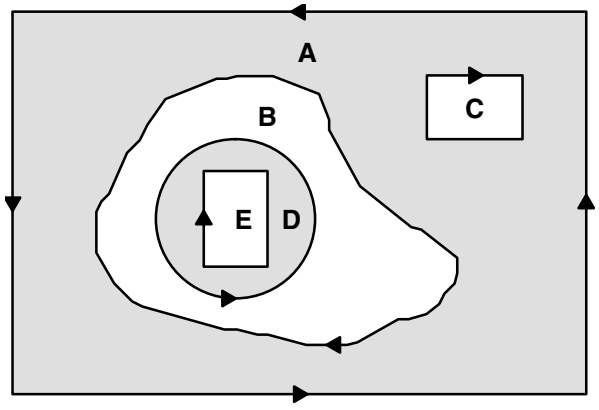
A trimming curve or trimming loop defines the visible regions in a NURBS surface. You can define trimming curves by the following methods:

- NURBS curves, using the **nurbscurve** subroutine.
- Piecewise linear curves, using the **pwlcurve** subroutine.
- Any combination of these two.

In any case, the trimming curve must be closed: that is, the coordinates of the first and last points of the trimming curve must be identical (within a tolerance of 10E-6). Because NURBS curves normally do not pass through the control point, one way to ensure a closed curve is to repeat the coordinates for the control point a number of times equal to the order of the curve. For example, quadruple the control points for a fourth-order curve if you wish to make the curve pass through that point. Another technique is to construct a knot vector that generates positional continuity of the endpoints of the curve.

When error checking is activated, the software sends error messages and does not display the NURBS surfaces associate with the faulty trim data. Likewise, the end points of piecewise linear curves and the NURBS curves used to form a compound trimming curve must touch.

A NURBS surface is the result of a mathematical function that maps domain space to model space. You determine the visible parts of the NURBS surface by defining a trim region. The trim region is the area of the NURBS surface in which the surface domain is trimmed by a closed directed loop (composed of one or more trimming curves) in s-t space, where the interior of the loop is defined to be the region to the left of the loop. The surface domain can be trimmed by many such loops, as long as they describe a consistent region. The loops can neither touch nor intersect (except at their end points, which must touch), and their orientations must also be consistent. The following figure illustrates a set of 5 loops that describe a valid trimming region. The image of the shaded portion is the trimmed NURBS surface.



Trimming Loops

If no trimming information is provided, the entire surface is drawn. If any trimming loops are given, the outer loop (or loops) must be counterclockwise. Thus to describe a region that consists of the whole surface minus a small circle in the middle, two trimming loops must be specified: one running clockwise around the circle, and another running counterclockwise around the entire s-t domain.

A trimming loop can be described either as piecewise linear curves (a series of s-t coordinates locating successive points along a path), or as NURBS curves in the s-t plane. A loop can be described either by a single NURBS curve, by a piecewise linear curve, or as a series of curves (of either type) joined head to tail.

The general form of the interface to describe a trimmed or untrimmed NURBS surface looks like this:

```
bgnsurface();
   nurbssurface(. . .);
bgntrim();
   nurbscurve(. . .);
endtrim();
bgntrim();
   pwlcurve(. . .);
endtrim();
bgntrim();
   nurbscurve(. . .);
   pwlcurve(. . .);
   nurbscurve(. . .);
endtrim();
endsurface();
```

Each trimming loop is surrounded by a **bgntrim** and **endtrim** subroutine pair. A single curve defines the first two trimming loops; the third loop consists of three segments, connected head to tail. The last point of each curve segment must touch the first point of the next, and the last point of the last segment must touch the first point of the first segment. The **nurbssurface** subroutine describes the untrimmed surface and appears before any trimming information. The trimmed surface description is bracketed by a **bgnsurface** and an **endsurface** subroutine pair.

The other subroutines specifically related to the properties of NURBS surfaces are **setnurbsproperty** and **getnurbsproperty**. These subroutines allow the user to set and get drawing tolerances of various types.

All the subroutines in the example, except for the **getnurbsproperty** subroutine, can be used in display lists. In this implementation, NURBS surfaces described in display lists usually run faster because some of the display computations can be cached between display list traversals.

All the parameters are passed with strict call-by-value semantics. This means that the system copies all values, including trim points and control points, at the time of the call. For example, if you have an array containing control points, and you define a NURBS surface in a display list using it and then change the value in your array, the display list will continue to draw the surface using the original control point values.

nurbscurve Subroutine

The **nurbscurve** subroutine can be used only within a **bgntrim/endtrim** loop and in curves of up to order 8.

The structure of the parameters is analogous to those for the **nurbssurface** subroutine, except, of course, there is only one dimension to describe. When the **nurbscurve** subroutine describes a trimming curve, it must be two-dimensional, so the only legal values for type are N_STW and N_ST. The control point formats for N_STW and N_ST are (*ws*, *wt*, *w*) and (*s*, *t*), respectively.

If a single curve defines the entire trimming loop, both ends of the curve must lie at the same point and must be included in the parameter count.

When you trim a NURBS surface with a NURBS trimming curve, the software analytically calculates coordinates on the surface and their corresponding normal vectors for each point on the tessellated NURBS trimming curve.

```
nurbscurve (
Int32 knot_count,   /* number of knots */
Float64 knot_list[],   /* non-decreasing knot sequence */
Int32 stride,      /* byte offset to next control point */
Float32 *ctlarray,   /* pointer to first control point */
Int32 order,      /* spline order */
Int32 type      /* spline type -- 2D, 3D, rational,
            polynomial */
)
```

pwlcurve Subroutine

To define a piecewise linear trimming curve, use the **pwlcurve** subroutine. The syntax is as follows:

```
void pwlcurve(Int32 count, Float64 *data_array,
              Int32 stride, Int32 type)
```

The trimming curve in the s-t plane is drawn by connecting each point in the *data_array* parameter to the next. It is as important to increment the trim point count as it is to duplicate the last point. In other words, although the last and first points are identical, they must be specified and counted twice.

Controlling Display Properties

The following subroutines control NURBS curves and surfaces display properties.

setnurbsproperty and getnurbsproperty Subroutines

The **setnurbsproperty** subroutine changes various properties that control the rendering of NURBS curves and surfaces. The call uses this format:

```
void setnurbsproperty(Int32 property, Float32 value)
```

A list of properties is defined in the **/usr/include/gl/gl.h** file and includes N_PIXEL_TOLERANCE and N_ERRORCHECKING. Each has some reasonable default value but can be changed to affect the accuracy of some part of the rendering. You can get the current value of any of these properties with a call to the **getnurbsproperty** subroutine. The syntax is as follows:

```
void getnurbsproperty(Int32 property, Float32 *value)
```

For maximum generality, express the value of a property in floating point. For some properties, only integer values make sense, but you must still pass them in floating-point form; for example, 1.0 means TRUE.

The values of the properties are global to a process, and each call to the **setnurbsproperty** subroutine changes this global state.

The properties have the following meanings:

N_PIXEL_TOLERANCE	A value representing how accurately a NURBS surface is to be rendered. Smaller values indicate more accuracy.
ERRORCHECKING	If TRUE, performs additional error checking.

Chapter 4. Working with Coordinate Systems

This section contains the following information:

- List of GL Coordinate Transformation Subroutines
- Coordinate Transformations
- User-defined Transformations
- Controlling the Order of Transformations
- Mathematical Details of the Matrix Subroutines

List of GL Coordinate Transformation Subroutines

getmatrix	Gets a copy of the current transformation matrix.
loadmatrix	Loads a transformation matrix.
lookat	Defines a viewing transformation.
mapw	Maps a point on the screen into line in 3-D world coordinates.
mapw2	Maps a point on the screen into line in 2-D world coordinates.
multmatrix	Premultiplies the current transformation matrix.
ortho	Defines a 3-D orthographic transformation.
ortho2	Defines a 2-D orthographic transformation.
perspective	Defines a perspective projection transformation in terms of a field of view.
polarview	Defines the viewer's position in polar coordinates.
popmatrix	Pops the transformation matrix stack.
pushmatrix	Pushes down the transformation matrix stack.
rot	Rotates a graphical primitive (floating-point version).
rotate	Rotates a graphical primitive (fixed-point version).
scale	Scales and mirrors graphical primitives.
translate	Translates a graphical primitive.
window	Defines a perspective projection transformation in terms of x and y coordinates.

Coordinate Transformations

When displaying 3-D shapes, it is useful to be able to move the shapes around relative to each other and to the viewer; to rotate and scale them; and to be able to change the viewer's point of view, field of view, and orientation. The subroutines that perform coordinate transformations allow you to manipulate geometric figures and viewpoints in 3-D space in very general ways.

GL converts the 3-D coordinates of geometric figures into pixels on the screen in the following operations:

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

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

Types of Coordinate Systems

There are basically five coordinate systems of interest. First, there is a 3-D system defined in right-handed Cartesian floating-point coordinates; vertices are specified in (*x,y,z*) triplets. Let us refer to this as the modeling coordinate system. There are no limits to the size of sensible coordinates (other than the largest legal floating-point value).

The second system is the world coordinate system, also a 3-D floating-point coordinate system. World coordinates are used conceptually for locating the entire scene. For example, the drawing of a bolt may be defined at the origin of the modeling coordinate system (because this is the easiest way to define a bolt), but that bolt may be drawn repetitively in many different places in world coordinates.

The third is called the eye, or viewer, coordinate system. The position of all things is measured with respect to the location of the viewer's eye. GL uses the same set of subroutines to manipulate the placement of shapes in these first three coordinate systems. These subroutines become the modeling, viewing, and projection transformations, depending on the order in which they are called and the mode the system is in.

The fourth is called the normalized coordinate system. This system is also 3-D with floating-point values, but its range is limited to -1.0 <= *x,y,z* <= 1.0. The 3-D cube defined by these limits is convenient for clipping. After transformation to the normalized system, the clipping hardware eliminates all geometry with coordinates outside the range of -1.0 to 1.0.

The *x* and *y* coordinates of this 3-D cube are scaled directly into the fifth coordinate system, usually called the screen coordinate system. If you draw into an arbitrarily placed window on the screen, the pixel at the lower-left corner of the window has screen coordinates (0,0). Because screen coordinates represent pixel values, they are always expressed in integers, so the transformation from normalized coordinates to screen coordinates might involve some rounding and consequent loss of accuracy.

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 many things, such as hidden surface removal and depth-cueing.

Types of Transformations

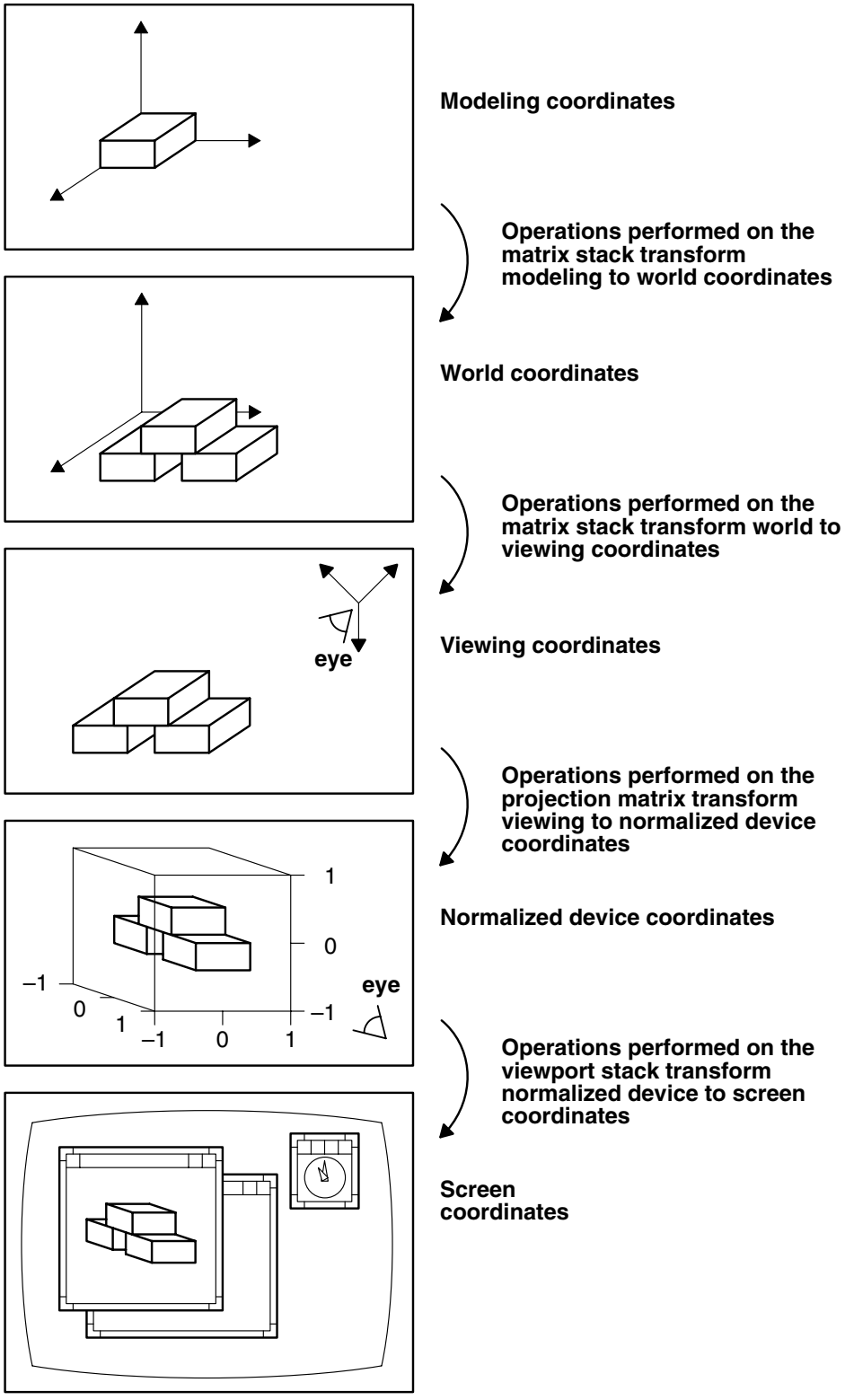
This section contains information on:

- Modeling Transformations
- Viewing Transformations
- Projection Transformations

To map between the five coordinate systems, there are four distinct types of coordinate transformations. These are as follows:

- Modeling transformations, which map from modeling coordinates to world coordinates. That is, they take a 3-D figure drawn in modeling coordinates and indicate how it is placed in world coordinates.
- Viewing transformations, which map from world coordinates to eye coordinates. That is, they indicate the location of the eye and the direction in which it is looking, and relate that to world coordinates.
- Projection transformations, which map from eye coordinates to normalized device coordinates. Projection transformations are usually used to control the amount of perspective in the scene.
- Viewport transformations, which map from normalized device coordinates (NDC) to device coordinates (DC) (also called screen coordinates or window-relative coordinates). These control the placement of the drawn scene on the monitor; that is, where it appears on the screen. Viewport transformations are not full-fledged transformations like the previous three; for example, rotations are not allowed. These transformations are controlled by the **viewport** and **lsetdepth** subroutines.

These transformations are represented in the following ″figure″ on page 83.



Coordinate Transformations

Modeling Transformations

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

All objects drawn after these subroutines execute are transformed as specified by the individual subroutine. Therefore, controlling the order in which you specify transformation operations is extremely important.

rotate Subroutine

The **rotate** subroutine rotates graphical objects by specifying an angle and an axis of rotation. The angle is given in tenths of degrees according to the right-hand rule: if the right hand is wrapped around the axis of rotation, the fingers curl in the same direction as positive rotation, and the thumb point in the same direction as the axis of rotation. A right-handed rotation is counterclockwise. An x, y, or z character defines the axis of rotation. (The character can be uppercase or lowercase.)

> Note: In the following discussion, the word *object* refers to the general idea of a drawn thing or shape (graphical primitive). It does not refer specifically to display lists.

All objects drawn after the **rotate** subroutine executes are rotated.

The **rot** subroutine is similar to the **rotate** subroutine, except that the angle is given in floating point. Both subroutines create a ″matrix″ on page 84 and premultiply it into the current matrix.

$$\mathrm{Rot}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{Rot}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{Rot}_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The **rot** and **rotate** subroutines syntax is as follows:

```
void rotate(Angle angle, Char8 axis)
void rot(Float32 angle, Char8 axis)
```

translate Subroutine

The **translate** subroutine moves the object origin to the point specified in the current object coordinate system. All objects drawn after the **translate** subroutine executes are translated. The **translate** subroutine creates a ″matrix″ on page 85 and premultiplies it into the current matrix.

$$\text{Translate } (T_x , T_y , T_z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

The syntax is as follows:

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

scale Subroutine

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

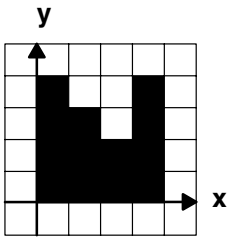
All objects that are drawn after the **scale** subroutine executes are scaled. The **scale** subroutine creates a ″matrix″ on page 85 and premultiplies it into the current matrix.

$$\text{Scale } (S_x , S_y , S_z) = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

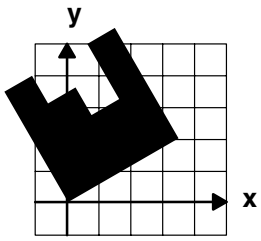
The syntax is as follows:

```
void scale(Float32 x, Float32 y, Float32 z)
```

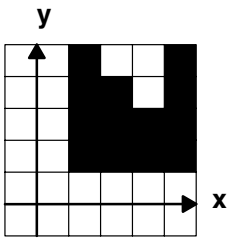
The modeling subroutines are illustrated in the following ″figure″ on page 86.



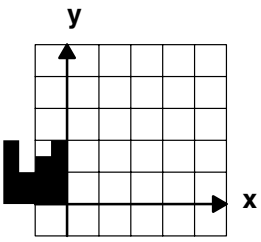
(a) original object at (0,0,0)



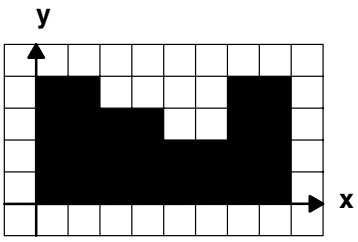
(b) rotate (300 ,'Z');



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



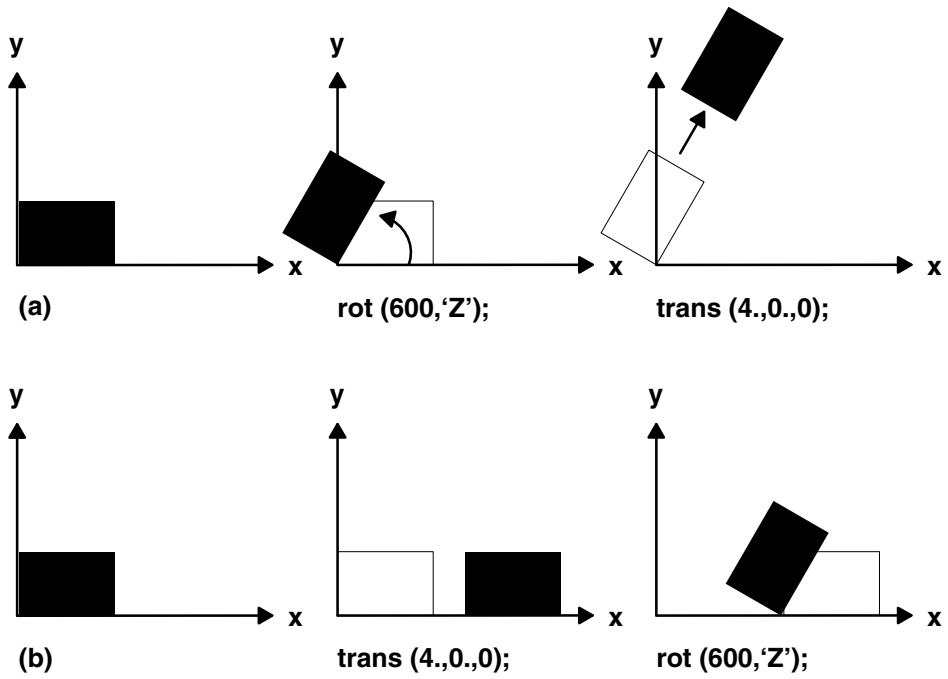
(d) scale (–.5,.5,1.);



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

Modeling Subroutines

The modeling subroutines are not commutative: if you reverse the order of operations, you can get different results. The following ″figure″ on page 87 shows (a) a rotation of 60 degrees about the origin followed by a translation of 4 degrees in the X direction. Part (b) shows the same operation performed in the reverse order. Rotations are about the origin of the coordinate system.



The translate and rotate Subroutines

Viewing Transformations

The viewing transformations allow you to specify the position of the eye in the world coordinate system, and to specify the direction toward which it is looking. The **polarview** and **lookat** subroutines provide convenient ways to do this.

polarview Subroutine

The **polarview** subroutine assumes that the object you are viewing is near the origin. The eye's position is specified by a radius (distance from the origin) and by angles measuring the azimuth and elevation. The specification is similar to polar coordinates, hence, the name. There is still one degree of freedom because these values tell only where the eye is relative to the object. A *twist* parameter tells which direction is up.

The angle of incidence equals the angle between the Z-axis in world coordinates and the location of the origin of viewing coordinates. The angle of azimuth equals the angle between the X-axis in world coordinates and the *x,y* coordinates of the origin of the viewing coordinates.

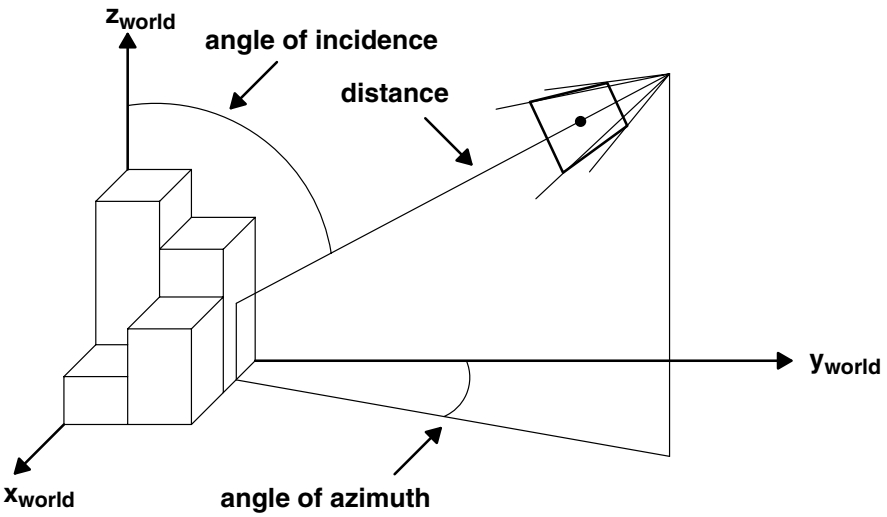
To understand incidence and azimuth, imagine that you are standing at the origin in world coordinates. You are facing north, along the Y-axis, with the X-axis on your right. The Z-axis points straight up, towards the zenith. There is a very large eye in the sky, looking down at you. It is located at the origin of the viewing coordinate system. This eye is the system, and whatever that eye sees appears on the screen.

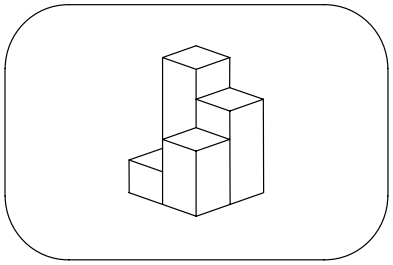
Where is the eye? The azimuth is the compass point at which it is located: 0 degrees if straight north, 90 degrees if straight east, and so on, in a clockwise fashion (in conformance with astronomical usage). The incidence is the angle down from zenith: 0 degrees means the eye is directly overhead; 90 degrees means that the eye is on the horizon.

The altitude (again following astronomical usage) is precisely equal to 90 degrees minus the incidence. This coordinate system is called *horizon coordinates* or *topocentric coordinates*. The syntax is as follows:

```
void polarview(Coord distance, Angle azimuth, Angle incidence,
      Angle twist)
```

The following ″figure″ on page 88 illustrates this viewpoint concept.

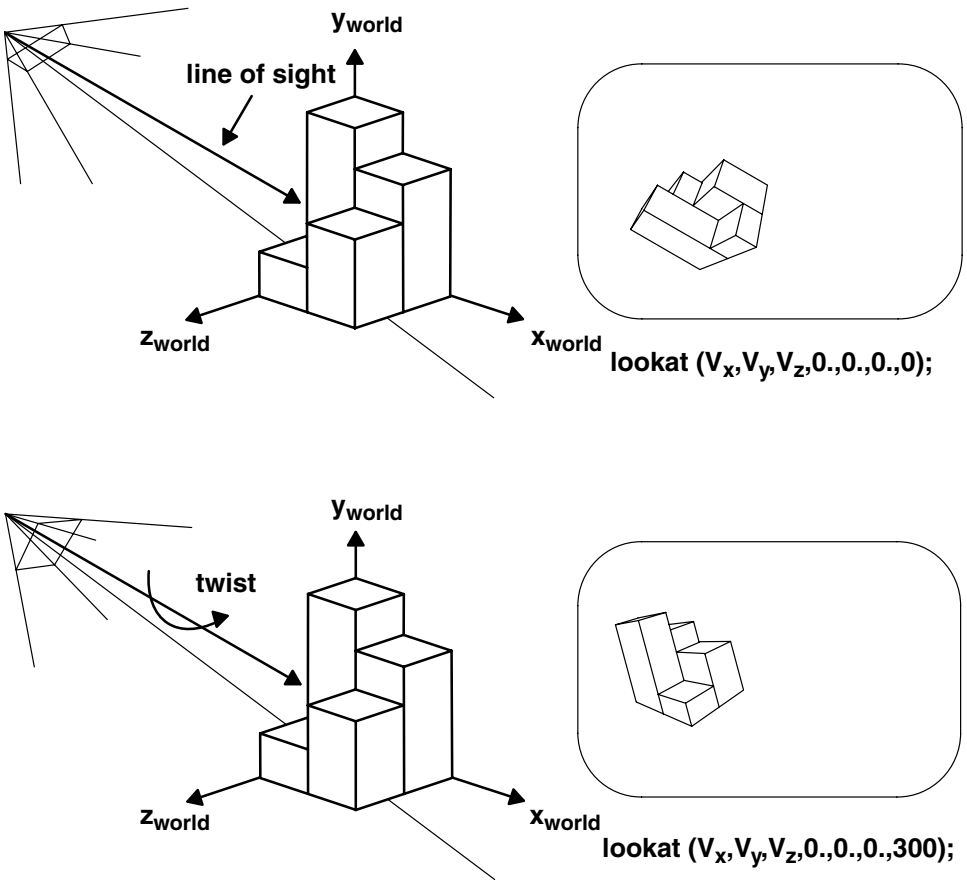




The polarview Subroutine

lookat Subroutine

The **lookat** subroutine allows you to specify the eye's position in space and a point at which it is looking. Both points are given with Cartesian *x*, *y*, and *z* coordinates. A *twist* parameter specifies the angle of rotation. Once you specify the eye position, the point you are looking at could be any point along a line, and the identical transformation is specified. This viewpoint concept is illustrated in the following ″figure″ on page 89.



The lookat Subroutine

Both viewing subroutines work in conjunction with a projection subroutine. If you wish to view point (1, 2, 3) from point (4, 5, 6) in perspective, use the **perspective** and **lookat** subroutines in conjunction. When the orthogonal projections are used, the exact position of the eye used in the viewing subroutines does not make any difference. The only thing that matters is the viewing direction.

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

The **polarview** and **lookat** subroutines create a ″matrix″ on page 90 and premultiply it into the current matrix.

Polarview (dist, azim, inc, twist) =
$[Rot_z(-azim)]\ [Rot_x(-inc)]\ [Rot_z(-twist)]\ [Trans(0.0, 0.0, -dist)]$

Lookat(v_x , v_y , v_z , p_y , p_y , p_y , twist) =
$[Trans(-v_x , -v_y , -v_z)]\ [Rot_y(\theta)]\ ]\ [Rot_x(\phi)]\ [Rot_z(-twist)]$

$$\text{where } \sin(\theta) = \frac{p_x - v_x}{\sqrt{(p_x - v_x)^2 + (p_z - v_z)^2}}$$

$$\cos(\theta) = \frac{v_z - p_z}{\sqrt{(p_x - v_x)^2 + (p_z - v_z)^2}}$$

$$\sin(\phi) = \frac{v_y - p_y}{\sqrt{(p_x - v_x)^2 + (p_y - v_y)^2 + (p_z - v_z)^2}}$$

$$\cos(\Phi) = \frac{\sqrt{(p_x - v_x)^2 + (p_z - v_z)^2}}{\sqrt{(p_x - v_x)^2 + (p_y - v_y)^2 + (p_z - v_z)^2}}$$

Projection Transformations

Viewing items in perspective on the computer screen is like looking at them through a rectangular piece of perfectly transparent glass. Imagine drawing a line from your eye through the glass to an item. The line colors a dot on the glass the same color as the spot on the item intersected by that line. If this were done for all possible lines through the glass, if the coloring were perfect, and the eye not allowed to move, then the picture painted on the glass would be indistinguishable from the true scene.

The collection of all the lines leaving your eye and passing through the glass would form an infinite four-sided pyramid with its apex at your eye. Anything outside the pyramid would not appear on the glass, so the four planes passing through your eye and the edges of the glass would clip out invisible items. These are called the *left*, *right*, *bottom*, and *top* clipping planes.

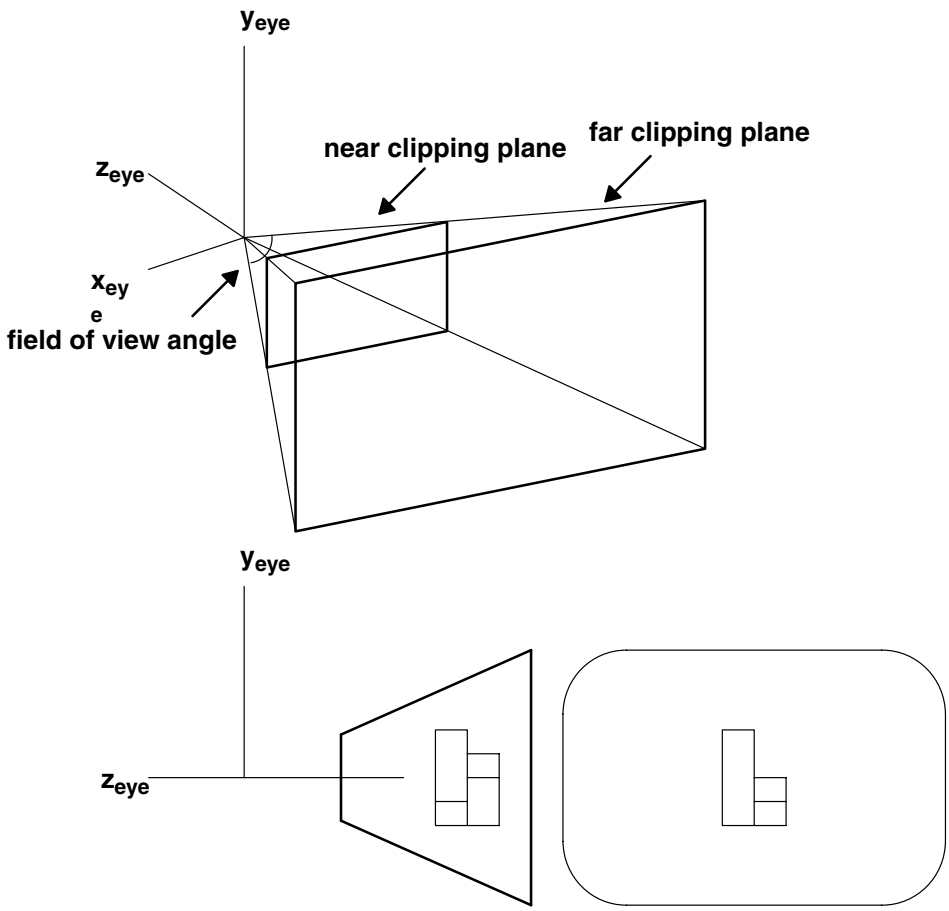
The geometry hardware also provides two other clipping planes that eliminate anything too far from the eye or too near the eye. They are called the *near* and *far* clipping planes. Near and far clipping is always turned on, but it is possible to set the near plane very close to the eye and the far plane very far from the eye so that all the geometric items of interest are visible.

Because floating-point calculations are not exact, it is a good idea to move the near plane as far as possible from the eye, and to bring in the far plane as close as possible. This gives optimal resolution for distance-based operations such as those discussed in Removing Hidden Surfaces and Performing Depth-Cueing.

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

perspective Subroutine

The **perspective** subroutine maps a frustum of eye, or viewer, space so that it exactly fills the viewport. This frustum is part of a pyramid whose apex is at the origin (0.0, 0.0, 0.0), whose base is parallel to the X-Y plane, and which extends along the negative Z axis. In other words, it is the view obtained when the eye at the origin looks down the negative Z axis, and the plate of glass is perpendicular to the line of sight, as shown in the ″figure″ on page 91.



The perspective Subroutine

The **perspective** subroutine has four parameters: the field of view in the y direction, the aspect ratio, and the distances to the near and far clipping planes. Typically, the aspect ratio is chosen 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.

Mathematically, the **perspective** subroutine works by mapping the 3-D volume enclosed by the viewing frustum into normalized device coordinates. Any point outside the frustum is mapped to a point outside the cube; that is, at least one of its coordinates is greater than 1.0 or less than -1.0. The clipping hardware then eliminates all the geometry outside the normalized viewing cube, and the *x* and *y* coordinates of the remaining geometry are scaled linearly to fill the window on the screen. The syntax is as follows:

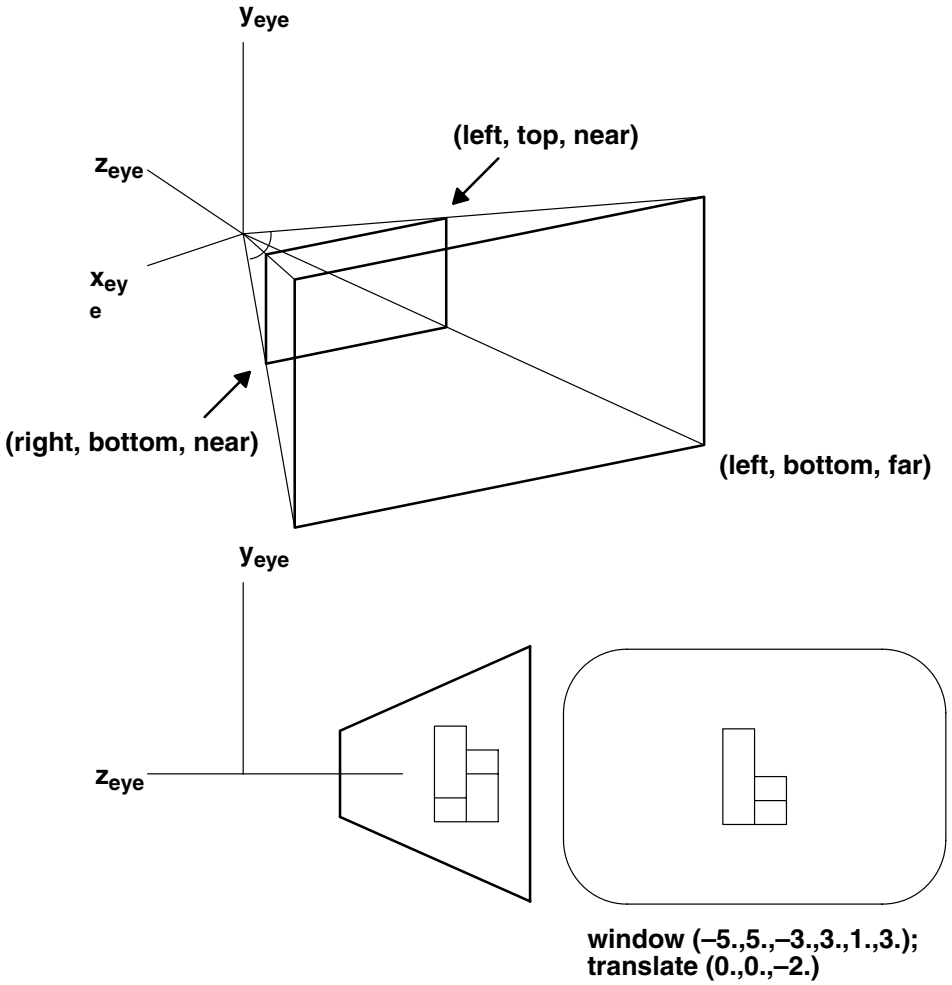
```
void perspective(Angle fovy, Float32 aspect,
      Coord near, Coord far)
```

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

window Subroutine

Another perspective projection transformation is the **window** subroutine. This subroutine is similar to the **perspective** subroutine, but its viewing frustum is defined in terms of distances to the left, right, bottom, top, near, and far clipping planes.

The **window** subroutine specifies the position and size of the rectangular viewing frustum closest to the eye (in the near clipping plane) and the location of the far clipping plane. The following ″figure″ on page 92 illustrates this function, defining a viewing window in the X-Y plane looking down the negative Z axis. A perspective view of the image is projected onto the window. The syntax is as follows:



The window Subroutine

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

The perspective transformation subroutines create ″matrices″ on page 93 and load them as the projection matrix.

Perspective
(fov, aspect, near, far) =

$$\begin{bmatrix} \dfrac{\cot\left[\dfrac{fov}{2}\right]}{aspect} & 0 & 0 & 0 \\ 0 & \cot\left[\dfrac{fov}{2}\right] & 0 & 0 \\ 0 & 0 & -\dfrac{far+near}{far-near} & -1 \\ 0 & 0 & -\dfrac{2xfarxnear}{far-near} & 0 \end{bmatrix}$$

Window
(left, right, bottom, top, near, far) =

$$\begin{bmatrix} \dfrac{2xnear}{right-left} & 0 & 0 & 0 \\ 0 & \dfrac{2xnear}{top-bottom} & 0 & 0 \\ \dfrac{right+left}{right-left} & \dfrac{top+bottom}{top-bottom} & -\dfrac{far+near}{far-near} & -1 \\ 0 & 0 & -\dfrac{2xfarxnear}{far-near} & 0 \end{bmatrix}$$

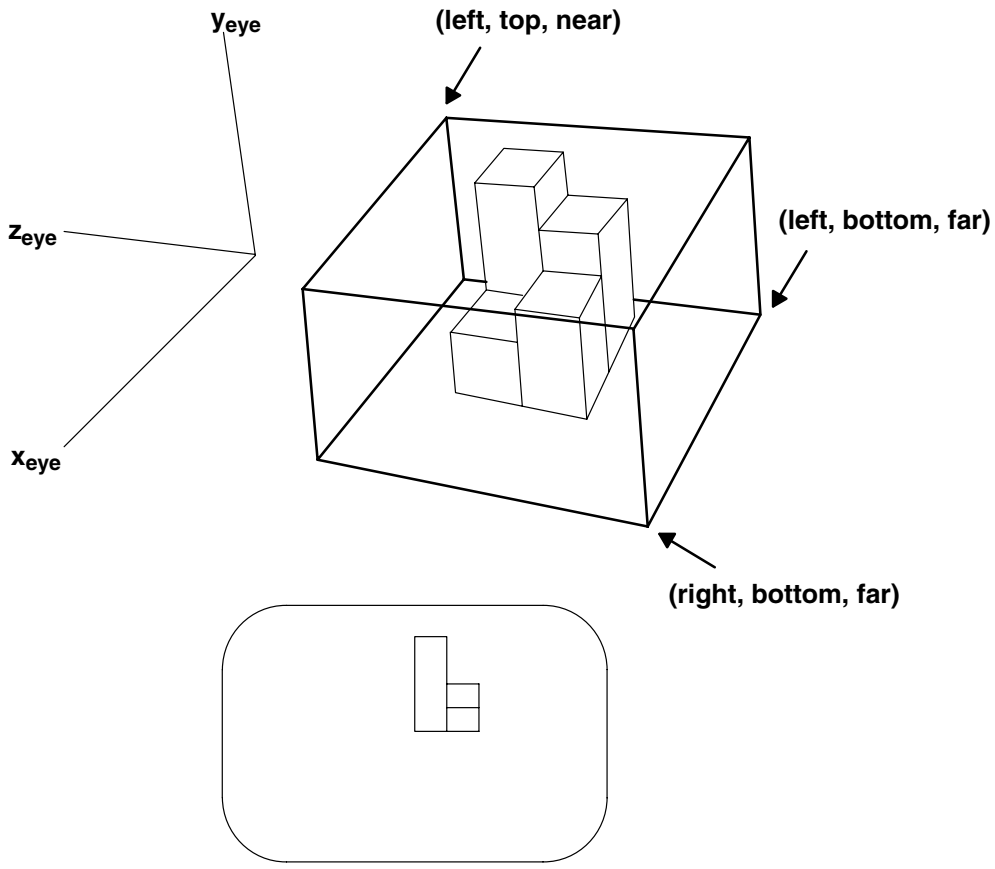
ortho and ortho2 Subroutines

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

Another way to think of the **ortho** subroutines is that the geometry outside the box is clipped out, and the geometry inside is projected parallel to the Z axis onto a face parallel to the X-Y plane.

The **ortho** subroutine allows you to specify the entire box: the X, Y, and Z limits. The **ortho2** subroutine, usually used for 2-D drawing, requires a specification of the X and Y limits only. The Z limits are assumed to be -1 and 1. Objects with *z* coordinates outside the range -1.0 <= z <= 1.0 are clipped out.

The following ″figure″ on page 94 illustrates this function, defining a viewing window in the X-Y plane looking down the negative Z axis. An orthographic view of the object between the near and far planes is projected onto the window. The syntax is as follows:



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

The ortho Subroutine

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

The orthographic subroutines create ″matrices″ on page 95 and load them as the projection matrix.

$$\text{Ortho}_{3d}(\text{left, right, bottom, top, near, far}) = \begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & 0 \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & 0 \\ 0 & 0 & -\frac{2}{\text{far}-\text{near}} & 0 \\ -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} & 1 \end{bmatrix}$$

$$\text{Ortho}_{2d}(\text{left, right, bottom, top}) = \begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & 0 \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} & 0 & 1 \end{bmatrix}$$

User-Defined Transformations

A transformation changes the size and orientation of an object by modifying either the object itself or the position of the viewpoint. A transformation is expressed as a 4x4 floating-point matrix. You can build complex transformations by linking a series of primitive transformation subroutines, such as **rotate**, **rot**, **translate**, or **scale**. If M,V, and P are modeling, viewing, and projection transformations, you can formulate transformation S, which maps model space into normalized device coordinates (NDC), as in the following equation:

S = MVP

[x y z w] M V P = [x' y' z' w']

The clipping boundaries are:

$x = \pm w$, $y = \pm w$, and $z = \pm w$

The resulting NDC coordinates:

$$\frac{x}{w}, \frac{y}{w}, \text{ and } \frac{z}{w}$$

are then scaled to the current viewport with the current viewport mapping.

The graphics pipeline maintains a stack that holds up to 32 transformation matrices. The system applies the matrix on top of the stack (the current transformation matrix) to all coordinate data.

The graphics pipeline forms a complex transformation matrix by premultiplying the current matrix by each primitive transformation. The pipeline forms transformation S by executing coordinate transformation subroutines in reverse order: first, projection subroutines; second, viewing subroutines; and third, modeling subroutines.

> **Note:** In MSINGLE matrix mode, the graphics pipeline loads the P transformation onto the matrix stack, while both the V and M transformations premultiply the current matrix. For additional information on matrix manipulation and the graphics pipeline, refer to Lighting in Matrix Mode.

The projection, viewing, and modeling subroutines provide a high-level interface that manages the transformation matrix stack. Additional subroutines allow direct control over the stack. These subroutines load or multiply user-defined transformation matrices, push and pop the transformation stack, and retrieve the matrix on the top of the stack.

loadmatrix Subroutine

The **loadmatrix** subroutine loads a 4x4 floating-point matrix onto the stack, replacing the current top of the stack. The syntax is as follows:

```
void loadmatrix(Matrix matrix)
```

multmatrix Subroutine

The **multmatrix** subroutine premultiplies the current top of the transformation stack by the given matrix. That is, if T is the current matrix, **multmatrix**(M) replaces T with MT. The syntax is as follows:

```
void multmatrix(Matrix matrix)
```

getmatrix Subroutine

The **getmatrix** subroutine copies the transformation matrix from the top of the transformation stack to an array provided by the user. The stack does not change. If lighting is not being used (the default case), the product MVP of the modeling, viewing, and projection matrices is kept on the stack. The syntax is as follows:

```
void getmatrix(Matrix matrix)
```

When lighting is being used, the projection matrix P is kept separately, and only the product of the modeling and viewing matrices, MV, is kept on the stack. A special mechanism, the matrix mode, is provided for accessing P and MV separately. Otherwise, the matrix subroutines work as previously described.

Establishing a One-to-One Mapping Between Screen Space and World Space

Specifying or defining the **ortho2** subroutine parameters brings up the issue of creating a window that has a one-to-one mapping between screen space (viewport) and world space (in this case, **ortho2**). Consider the following example.

Assume a window that is four pixels wide by six pixels high. This window runs from coordinates 0 to 3 on the X-axis and from 0 to 5 on the Y-axis. In order to set up a mapping between world space (floating-point coordinates) and screen space (integer coordinates) that makes pixel (1,2) centered exactly at the point (1.0, 2.0) in the **ortho2** world space, you must call the following subroutines:

```
viewport(0, 3, 0, 5);
ortho2(-0.5, 3.5, -0.5, 5.5);
```

To understand why these values are correct, consider the X component. The width in X of this window is 4 pixels, which are integer values; it makes no sense to talk about pixel 1.3. In world coordinates, however, an X location of 1.3 is valid. The mapping from world to screen coordinates attempts to convert the X world coordinate 1.3 to the nearest whole-number pixel box it can find. Rounding off 1.3 points GL at pixel 1.

The call to the **ortho2** subroutine runs between *x* values of 0.5 and 3.5 in order to let the rounding operation center the four *x* world-space whole-number values of 0.0, 1.0, 2.0, and 3.0 in the middle of each pixel in the X dimension.

In this scheme, -0.5 can be thought of as the extreme left-hand edge of the window, while 3.5 is the extreme right-hand edge, 1.5 is the boundary between pixel 1 and pixel 2, and so on. This lets you define the *x* range in the **ortho2** subroutine so that, in effect, the world coordinates straddle the discrete whole number boundaries and center the whole numbers (0.0, 1.0, 2.0, 3.0) in the middle of each pixel (0, 1, 2, 3).

Extrapolate from this and assume a situation where the window has been resized and you need to redefine a current **ortho2** subroutine based on the new size. To do this, use the following three statements:

```
getsize(&xsize, &ysize);
viewport(0, xsize - 1, 0, ysize - 1);
ortho2  (-0.5,(float)(xsize-0.5),
         -0.5,(float)(ysize-0.5);
```

In the call to the **viewport** subroutine, you must subtract 1 from the value of `xsize` and `ysize` because they start at zero, not one. Likewise, in the call to the **ortho2** subroutine, you need to start at 0.5; therefore, you need to subtract 0.5 from `xsize` and `ysize` to create the straddling effect described previously.

Controlling the Order of Transformations

Each time you specify a transformation such as rotation or translation, the software automatically generates a transformation matrix that specifies the amount by which the object's coordinate system is to be premultiplied. This transformation matrix is loaded atop a hardware stack in the graphics pipeline. When you specify a series of transformations, the software loads each successive transformation matrix onto the stack and modifies the current transformation matrix.

Because the graphics pipeline maintains a stack of matrices, you can save transformation matrices that define a particular state by manipulating the matrix stack.

Hierarchical Drawing with the Matrix Stack

A drawing is often composed of copies of simpler drawings, each of which may itself be composed of still simpler drawings. For example, if you were writing a program to draw a picture of a bicycle, you might want to have a subroutine that draws a wheel. Calling that subroutine twice, appropriately translated, would draw two wheels. 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 could 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 relative to its axis). You would then remove the forward translation to get back to the bicycle's frame of reference, translate back, and draw another instance of the wheel.

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

This is easily done using the matrix stack. At any point in a drawing, there is a current matrix on top of the matrix stack, composed of all the transformations thus far. In the bicycle example, we call this collection of transformations M. Any vertex is transformed by the top matrix, which is just what you want to do for drawing the frame.

The **pushmatrix** and **popmatrix** subroutines push and pop the matrix stack. The **pushmatrix** subroutine pushes the matrix stack down and copies the current matrix to the new top. Following a call to the **pushmatrix** subroutine, there would be two copies of M on top. A call to the **translate** subroutine (by a translation matrix S) leaves the stack with SM on top and M underneath. The wheel is then drawn once using the SM transformation. A call to the **popmatrix** subroutine then eliminates the SM on top, leaving M. A second call to the **pushmatrix** subroutine makes two copies of M.

Translating by T puts TM on top, so you can now draw the back wheel. After the matrix stack is popped again, M is on top, and you can draw the rest of the frame. The code for drawing the bicycle would look similar to this:

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

As shown in the code, the **drawwheel** subroutine can itself push and pop matrices before calling the **drawspoke** subroutine.

pushmatrix Subroutine

The **pushmatrix** subroutine pushes down the transformation stack, duplicating the current matrix. For example, if the transformation stack contains one matrix, M, then after a call to the **pushmatrix** subroutine, it contains two copies of M. You can modify only the top copy. The syntax is as follows:

```
void pushmatrix()
```

popmatrix Subroutine

The **popmatrix** subroutine pops the transformation stack. When this subroutine executes, the matrix on top of the stack is lost. The syntax is as follows:

```
void popmatrix()
```

Mathematical Details of the Matrix Subroutines

This section defines the formulas used by the **rot**, **rotate**, **translate**, **scale**, **lookat**, **polarview**, **window**, and **perspective** subroutines. These formulas are placed here for information only. It is not necessary to understand their meaning to use them successfully.

The **rot** and **rotate** subroutines create the following ″matrix″ on page 84 and premultiply it into the current matrix.

The **translate** subroutine creates the following ″matrix″ on page 85 and premultiplies it into the current matrix.

The **scale** subroutine creates the following ″matrix″ on page 85 and premultiplies it into the current matrix.

The **polarview** and **lookat** subroutines create the following ″matrix″ on page 90 and premultiply it into the current matrix.

The perspective transformation subroutines create the following ″matrices″ on page 93 and load them as the projection matrix.

The orthographic subroutines create the following ″matrices″ on page 95 and load them as the projection matrix.

Chapter 5. Using Viewports and Screenmasks

The viewport is the area of the window that displays an image. You specify it in window coordinates, where the coordinates of the lower left corner of the window are (0, 0). A more technically accurate definition of the viewport is that it is the mapping from normalized device coordinates (NDC), the coordinate frame in which the 3-D clipping is performed, to window coordinates.

Because 3-D clipping occurs in NDC, no drawing primitive is ever drawn outside the boundaries of the unit cube in NDC. Thus, when the unit cube of the NDC is transformed to window coordinates, no drawing primitive draws outside the viewport. An exception to this are text strings. Refer to Creating Text Characters for more information.

For more information on using viewports and screenmasks in GL, see:

- List of GL Viewport and Screenmask Subroutines
- viewport Subroutine
- getviewport Subroutine
- scrmask Subroutine
- getscrmask Subroutine
- pushviewport Subroutine
- popviewport Subroutine

List of GL Viewport and Screenmask Subroutines

getscrmask	Returns the current screenmask.
getviewport	Gets a copy of the dimensions of the current viewport.
lgetdepth	Gets the distance of the near and far clipping planes.
lsetdepth	Sets the viewport depth range.
popviewport	Pops the viewport stack.
pushviewport	Pushes the viewport onto the viewport stack.
reshapeviewport	Sets the viewport to the dimensions of the current window.
screenspace	Interprets graphics positions as absolute screen coordinates.
scrmask	Defines a rectangular 2-D clipping mask.
viewport	Set the area of the window used for all drawing.

viewport Subroutine

The **viewport** subroutine specifies, in window coordinates, the area of the window that displays an image. By default, when a window is opened on the screen, its viewport is set to cover the whole window. Its parameters (*left*, *right*, *bottom*, *top*) define a rectangular area on the window by specifying the left, right, bottom, and top coordinates. The portion of modeling space that the **window**, **ortho**, or **perspective** subroutine describes is mapped into the viewport. The syntax is as follows:

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

getviewport Subroutine

The **getviewport** subroutine returns the current viewport. Its parameters (l*eft*, *right*, *bottom*, *top*) are the addresses of four memory locations. These are assigned the left, right, bottom, and top coordinates of the current viewport. The syntax is as follows:

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

scrmask Subroutine

The **scrmask** subroutine specifies a two-dimensional rectangular clipping mask. The screenmask is specified in window coordinates. Every drawing primitive is clipped to this area: that is, no drawing primitive can draw outside the screenmask. The clipping performed by the screenmask is separate from and independent of the 3-D clipping performed by the system. The syntax is as follows:

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

Normally, the screenmask is precisely the same size as the window. The screenmask cannot be made larger than the window. When the **viewport** subroutine is called, it resets the screenmask to be the same size as the viewport.

Although the **scrmask** subroutine clips all primitives, it is not useful except for performing character clipping. The 3-D clipping performed by the system is sufficient for all other clipping. However, in reference to character strings, the 3-D clipping mechanism only clips the origin of the character string. That is, if the origin of the character string is inside the viewport, the entire string would be drawn if the screenmask was not in effect. If the origin is outside the viewport, none of the character string would be drawn, even if, logically, some of the string ought to be visible.

This type of clipping is called gross clipping. The screenmask is provided for fine clipping, clipping down to a subcharacter level. To use it, set the screenmask to be smaller than the viewport. Strings that begin inside the viewport are drawn, but only starting at the character (or fraction of the character) that lies inside the screenmask. Please refer to Creating Text Characters for more information.

getscrmask Subroutine

The **getscrmask** subroutine returns the coordinates of the current screenmask in the parameters *left*, *right*, *bottom*, and *top*. The syntax is as follows:

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

pushviewport Subroutine

The system maintains a stack of viewports and screenmasks, and the top element in the stack is the current viewport and screenmask. The **pushviewport** subroutine duplicates the current viewport and screenmask and pushes this duplicate element on the stack. The syntax is as follows:

```
void pushviewport()
```

popviewport Subroutine

The **popviewport** subroutine pops the stack of viewports and screenmasks. The viewport and screenmask element on top of the stack is lost. The new top element becomes the current viewport and screenmask. The syntax is as follows:

```
void pushviewport()
```

Chapter 6. Removing Hidden Surfaces

This section discusses the following aspects of hidden surface removal:

- List of GL Hidden Surface Removal Subroutines
- Understanding Hidden Surface Removal
- Backfacing Polygon Removal
- Z-Buffering
- Control of Z Values
- Additional Z-Buffer Features, including reading the z-buffer, hidden surface removal in the overlay planes, drawing into the z-buffer, alternate comparisons, and z-buffer writemasks

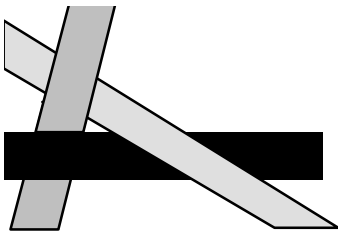
List of GL Hidden Surface Removal Subroutines

backface	Allows or suppresses display of backfacing polygons.
frontface	Controls frontfacing polygon removal.
getbackface	Indicates whether backfacing polygon removal is on or off.
getzbuffer	Indicates whether z-buffering is on or off.
zbuffer	Enables or disables the z-buffer for storing depth information.
zdraw	Enables drawing to the z-buffer.
zfunction	Specifies the function used for depth comparison.
zsource	Selects depth or color as the source for *z* comparisons.
zwritemask	Specifies which bits of the z-buffer are drawn during normal z-buffer operation.

Understanding Hidden Surface Removal

By default, GL does no hidden surface removal: figures are rendered on the screen in the order that they are drawn. For most 3-D drawings, it is important to draw only those surfaces that are nearest the eye, at least for opaque objects. All other surfaces are obscured by those nearer the eye. The time it takes to draw surfaces that are not visible in the final scene can cause a noticeable degradation of performance in complex scenes where drawing speed is important.

There are many ways to do hidden surface removal, depending on the types of figures being rendered. The primary method used by GL is a z-buffer, which works by calculating the distance to the eye from the surfaces covering each pixel and drawing only the closest surface. The calculation has to be done on a per-pixel basis because it is possible to have a set consisting of as few as three polygons, each of which is overlapped by another in the set, as in the following figure:



Overlapping Polygons

Note: Z-buffer hardware is optional on the High Performance 8-Bit 3-D Color Graphics Processor and the High Performance 24-Bit 3-D Color Graphics Processor. Be sure that you have a z-buffer installed on your system before using z-buffer features. Z-buffering does not work unless the z-buffer option is installed.

Backfacing Polygon Removal

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

A backfacing polygon is defined as a polygon whose vertices appear in clockwise order in screen space. When backfacing polygon removal is turned on, the only polygons displayed are those in which the vertices appear in counterclockwise order, that is, polygons that point toward you. Therefore, all polygon vertices should be specified in counterclockwise order.

backface Subroutine

The **backface** subroutine initiates or terminates backfacing polygon removal. It is used to improve the performance of programs that represent solid shapes as collections of polygons. The vertices of the polygons on the far side of the solid are in clockwise order and are not drawn.

The **backface** subroutine takes a single parameter. A value of True enables backfacing polygon elimination and False (the default) disables it. The syntax is as follows:

```
void backface(Int32 bool)
```

getbackface Subroutine

The **getbackface** subroutine returns the state of backfacing filled polygon removal: 1 if removal is enabled, 0 if disabled.

The backfacing polygon removal is not as general as z-buffering. However, it can be useful for drawing polygons whose front side is drawn with a different color and pattern than the back side. (The polygon would have to be drawn twice, with the vertices in reversed order the second time.) The syntax is as follows:

```
Int32 getbackface()
```

Z-Buffering

The z-buffer determines which of the things being drawn (polygons, lines, or text) is closest to the viewer. The distance to the figure being drawn is compared to the distance of the shapes already drawn in the scene. If the current figure is more distant (and therefore hidden), it is not drawn.

The z-buffer is a set of 24-bit integers, one associated with each pixel of the screen. You start by clearing the z-buffer: that is, by setting the *z* value of each pixel to the most distant (most positive) value possible. Then as each polygon, line, point, or character is rendered, its *x* and *y* screen coordinates are calculated in the usual way. Before the pixel is colored, however, the *z* coordinate is also calculated. The *z* coordinate is effectively the distance to the eye.

This incoming *z* value is compared to the z-buffer value already stored for that pixel. If the *z* value is smaller than the value in the z-buffer, the pixel is colored, and the pixel's z-buffer value is set to the new *z*

value. At any point in the drawing, the values in the z-buffer represent the distance to the item that is currently closest to the eye. The color value stored in the bitplanes represents the color of that item. The *z* comparison is unsigned.

Near and far 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 fixed and finite number of integer values that can be used to compare against the *z* value of the object in the scene. With this capability, you can control the resolution of the z-buffer by setting the near and far values. The more closely you bracket the objects between the near and far clipping planes, the better *z* compare resolution you achieve.

The **zbuffer2.c** example program draws three cubical objects (they are all originally perfect cubes, but scale stretches them along their axes). The objects tumble through each other and the whole scene is also rotating. While the left mouse button is up, the scene is drawn without z-buffering. When it is down, z-buffering is enabled. If the program is called with an argument, there is a short delay between drawing each of the polygons. In this mode, the left mouse button still controls the z-buffering.

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

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

The first subroutine enables z-buffer comparisons to be made before each write, and the second sets all the *z* values to the largest possible value for pixels in the viewport. In this example, `zbuffer(TRUE)` is called for every frame, but this is not necessary. The `zbuffer(TRUE)` call in a typical program is called only at the beginning. In the **zbuffer2.c** example program, the code is written as it is because the left mouse button can come up at any time, in which case z-buffering should be turned off.

The **getzbuffer** subroutine returns True or False, depending on whether z-buffering is enabled or not. By default, z-buffering is turned off; most applications that require 3-D hidden surface removal should probably turn on z-buffering.

Control of Z Values

When coordinate data is transformed to screen coordinates by the graphics pipeline, the transformation is done in two steps. First, the data is multiplied by a transformation matrix that transforms all visible vertices so that they lie in the 3-D cube -1.0 <= *x*, *y*, *z* <= 1.0. The clipping hardware eliminates all data outside this cube. The second stage of the transformation scales the coordinate values (now transformed to lie between -1.0 and 1.0) to screen coordinates.

The *x* and *y* coordinates undergo scaling to the viewport coordinates before they are drawn, and the *z* coordinates undergo the same transformation. The *x*, *y* scaling is controlled by the **viewport** subroutine, and the *z* scaling by the **lsetdepth** subroutine.

In the default mode with z-buffering turned off, the scaled *z* coordinates are ignored, and the drawing depends only on the *x* and *y* coordinates. When z-buffering is turned on, the *z* values are compared to determine whether to draw each pixel. The *z* values for these comparisons are based on the scaled *z* values.

> Note: The preferred programming practice to determine the minimum and maximum z values that a given hardware platform supports is to use the **getgdesc** subroutine with the GD_ZMIN and GD_ZMAX tokens.

lsetdepth Subroutine

The **lsetdepth** subroutine controls the scaling of the *z* coordinates, just as the **viewport** subroutine controls the scaling of the *x* and *y* coordinates. The **lsetdepth** subroutine takes two parameters of type Int32, corresponding to the near and far planes. By default, the near parameter is set to the minimum

value that can be stored in the z-buffer and the far parameter is set to the maximum value. The minimum value is -0x800000, the maximum value is +0x7fffff, the largest and smallest values that can be written into a 24-bit z-buffer. The syntax is as follows:

```
void lsetdepth(Int32 near, Int32 far)
```

czclear Subroutine

A very common code sequence in programs that do z-buffering is the following:

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

This sequence clears the color bitplanes to zero and clears the z-buffer bitplanes to the maximum value. Execution of the sequence takes time because the **clear** subroutine touches each pixel, and then the **zclear** subroutine touches each pixel. Some hardware implementations can, in certain cases, simultaneously clear the color planes and the z-buffer planes. The **czclear** subroutine allows you to do this.

The **czclear** subroutine clears the bitplanes to color and the z-buffer to the value of the *zval* parameter simultaneously. This subroutine is available only in immediate mode.

To speed up the **czclear** subroutine by as much as a factor of four for common values of the *zval* parameter, call the **zfunction** subroutine in conjunction with it. One of the following conditions must be met:

Conditions	
zval	**zfunction**
-0x800000	**ZF_GREATER or ZF_GEQUAL**
+0x7FFFFF	**ZF_LESS or ZF_LEQUAL**

The syntax is as follows:

```
void czclear(Int32 cval, Int32 zval)
```

Additional Z-Buffer Features

This section deals with special features that control z-buffering. Topics covered include reading the z-buffer, using the z-buffer for hidden surface removal in overlay planes, drawing directly into the z-buffer, using alternate depth comparison functions and sources, and writemasks for the z-buffer.

Reading the Z-Buffer

The contents of the z-buffer can be obtained by using the **zdraw** subroutine in conjunction with the **lrectread** subroutine.

Due to the nature of the design of the 3D Color Graphics Processor, values read back from the z-buffer may appear to be incorrect. To avoid obtaining peculiar z values, use the **czclear** subroutine to clear the z-buffer before drawing, and specify any z-value other than 0x7fffff or -0x800000 for the clear value. For instance, clearing to a value of -0x7ffffe can work without sacrificing dynamic range or accuracy. Note, however, that clearing to such non-standard z-values may degrade z clear performance and affect overall application performance.

Note: Not all graphics adapters support reading the z-buffer.

Hidden Surface Removal in the Overlay Planes

The z-buffer can be used in conjunction with the overlay planes to provide hidden-line and hidden-surface removal in the overlays. In a typical application, a solid-shaded figure might be drawn in the main frame buffer, while a wire-frame drawing is actively dragged about in the overlays. If the wire-frame drawing is dragged behind the solid-shaded figure, the z-buffer automatically prevents the hidden portion of the wire-frame drawing from being drawn. When performing such dragging, be sure to enable the z-buffer (by setting the z-buffer to True), and to set the z-buffer writemask so that the wire-frame drawing does not change any values already in the z-buffer.

> **Note:** Unless the above effect is specifically desired, it is recommended that z-buffering be disabled (by setting the z-buffer to False) while drawing into the overlay planes. Otherwise, confusing visual images may be the unintentional result when a drawing in the overlay becomes z-buffered.

Drawing into the Z-buffer

In some cases it is useful to be able to draw directly into the z-buffer. For instance, the z-buffer can be used as a clipping mask by writing near values into the region to be masked. If no other primitive is nearer than the value written into the z-buffer, the z-buffer hardware prevents the pixels from being written, effectively masking the region.

On the High Performance 3-D Color Graphics Processor, the **zdraw** subroutine enables block transfer of pixels into and out of the z-buffer. If the **zdraw** subroutine is enabled, the **lrectwrite** and **lrectread** subroutines transfer pixels to and from the z-buffer.

The **zdraw** subroutine is similar to the **frontbuffer** and **backbuffer** subroutines 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 and back buffer at the same time. Usually, drawing into the z-buffer should be bracketed by subroutines that set `backbuffer(FALSE)` and then `backbuffer(TRUE)` afterwards (assuming the program is in double buffer mode).

In single buffer mode, the **frontbuffer** subroutine normally has no effect. However, if you call `frontbuffer(FALSE)`, a flag is set so that when the **zdraw** subroutine is set to `TRUE`, the front buffer (the only buffer in single buffer mode) is not written into. If the **zdraw** subroutine is set to `FALSE`, `frontbuffer(FALSE)` has no effect.

Alternate Comparisons

In the default mode, the *z* coordinate of the new pixel is compared to the *z* coordinate of the figure currently 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.

The new value is compared to the old, and if it is less than the old, the old quantities are replaced. It is possible to change the comparison function from less-than to many other things. The available comparisons are shown in the following list.

Comparison Function	Definition
ZF_NEVER	Never overwrite the source pixel value.
ZF_LESS	Overwrite the source pixel value if the *z* of the source pixel value is less than the z of the destination value.
ZF_EQUAL	Overwrite the source pixel value if the z of the source pixel value is equal to the z of the destination value.
ZF_LEQUAL	Overwrite the source pixel value if the z of the source pixel value is less than or equal to the z of the destination value (default).
ZF_GREATER	Overwrite the source pixel value if the z of the source pixel value is greater than the z of the destination value.
ZF_NOTEQUAL	Overwrite the source pixel value if the z of the source pixel value is not equal to the z of the destination value.

Comparison Function	Definition
ZF_GEQUAL	Overwrite the source pixel value if the z of the source pixel value is greater than or equal to the z of the destination value.
ZF_ALWAYS	Always overwrite the source pixel value regardless of destination value.

To change the comparison function, use the **zfunction** subroutine.

Z-buffer Writemask

The **zwritemask** subroutine can be used like other writemasks to control writing into the z-buffer. The two valid settings are `0` (no write at all) and `0xffffff` (write all the bits). This subroutine could be useful for a very complicated background into which a few objects are going to be drawn and moved quickly. Setting the **zwritemask** to `0` locks the background information in and prevents its modification. New objects are drawn or not depending on whether the depth comparison indicates that they are in front of or behind anything else in the scene.

Chapter 7. Creating Lighting Effects

This section discusses the following topics:

- List of GL Lighting Subroutines
- Lighting Introduction
- Lighting Basics
- Advanced Lighting Capabilities
- Lighting in Matrix Mode
- Lighting Subroutines
- Lighting Execution Time and Performance
- Formula for Lighting Calculation

List of GL Lighting Subroutines

getmmode	Returns the current matrix mode.
lmbind	Binds a new material, light source, or lighting model definition.
lmcolor	Respecifies the currently bound material properties.
lmdef	Defines a new material, light, or lighting model.
mmode	Sets the current matrix mode.

Lighting Introduction

The GL lighting model facility automatically calculates color using the geometries, colors, and properties of the currently bound material, lights, and lighting model. The lighting model subroutines can define multiple materials, lights, and lighting models. A lighting equation consists of one material, eight lights, and one lighting model.

A significant feature of GL is that the hardware can efficiently perform the lighting calculations needed to enhance the realism of the displayed geometry. Because different applications require different degrees of realism, the lighting facility in GL provides a variety of capabilities and features that allow you to control the degree of realism in the images produced by your application. You can use many of these realism features without a negative effect on the drawing performance.

When you look at an object in the real world, the color of that object (as perceived by your eye) depends on several parameters:

- The color, location, and direction of the light source or sources illuminating the object.
- The color and surface properties of the object itself.
- The position and view direction of the observer.

Although GL allows you to control all of these parameters, you will find that you can produce quite realistic images by using only a subset of the available parameters.

Lighting Basics

This section discusses the basic techniques of lighting, including:

- A Simple Lighting Calculation
- Specularity
- Multiple Surface Materials and Multiple Lights

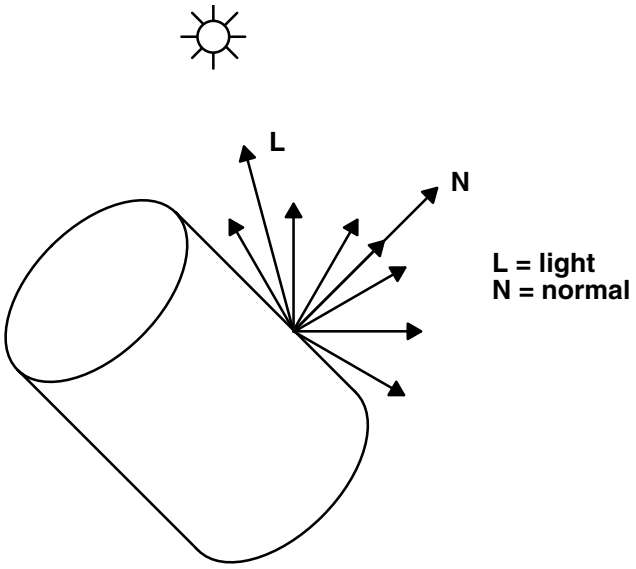
A Simple Lighting Calculation

The **cylinder1.c** example program (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) illustrates how to use GL to light a cylinder using a simple lighting calculation. Consider a pale gray cylinder with a rough surface that is illuminated by a source of white light. Objects with rough surfaces tend to reflect or scatter light equally in all directions. Such reflections are termed diffuse reflections.

In our simple lighting calculation, the color and intensity of the diffusely reflected light at a point on the surface depends on three things:

- Light source color
- The angular relationship between the light source direction and reflecting surface orientation at that point
- The reflectance characteristics of the cylinder surface

The short arrows in the following ″figure″ on page 110 show how the diffuse light is scattered equally in all directions.



Diffuse Lighting

The orientation of the reflecting surface is specified by the surface normal vector, which is perpendicular to the surface at the point in question. The normal vector must be normalized (must be of unit length). Because the surface normal differs at every point on a curved surface, the calculation for reflected light produces a different color at every point on the surface.

The number of points used in the lighting calculations depends on the number of vertices used to define the surface. For a local light (a light not very far from the surface), the system computes a color at every vertex of the polygon mesh used to represent the surface. The application writer must specify the surface-normal vector at every vertex of the polygon mesh.

In addition to the light originating from the light source that reflects off the surface, ambient light is also accounted for in the model. Ambient light is assumed to be nondirectional and is reflected uniformly in all directions by the reflecting surface. Thus, the color and intensity of the reflected ambient light are functions of both the level of ambient light in the scene and the reflectance properties of the surface. They are not functions of the angular relationship between the source of light and the surface normal.

Subroutines from Example Program cylinder1.c

The example C language program **cylinder1.c** introduces four new subroutines that use the GL lighting facility. The new subroutines are **mmode**, **lmdef**, **lmbind**, and **n3f**.

To perform lighting calculations, the system must be able to distinguish between the projection matrix (used by the **perspective** subroutine) and the viewing and modeling matrix (used by the **lookat** subroutine and the **rotate** subroutine). The **mmode** subroutine is used to indicate to the hardware which matrices represent projection transformations and which represent viewing or modeling transformations.

The **lmdef** subroutine, called from the `def_simple_light_calc` subroutine of the example program, found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*, defines instances of the three necessary components to perform a lighting calculation:

- A surface material
- A light source
- A lighting model

The program line `lmdef(DEFMATERIAL, 1, 0, NULL)` tells GL about surface material number 1. The value given the first parameter, `DEFMATERIAL`, indicates that you are defining a surface material. The value for the second parameter defines this material as material number `1`. The value for the third parameter (`0`) is the length of the properties array specified by the fourth parameter. The properties array specifies the properties of the surface material. Here, NULL instructs GL to use the default values for surface materials.

The next two calls to the **lmdef** subroutine are similar to the first. By substituting `DEFMATERIAL` with `DEFLIGHT`, and subsequently `DEFLMODEL`, you can define a light source and lighting model using the default values prescribed by GL. The default values for a surface material, a light source, and a lighting model are all that are necessary to perform a simple lighting calculation. Properties of a lighting model are the properties that affect the entire scene rather than a specific light source or surface material (the intensity and color of the ambient light in the scene). The subroutine **lmdef** is covered in greater detail in the section on lighting subroutines.

Another subroutine introduced in the example program is the **lmbind** subroutine. Once you have defined a surface material, light source, and lighting model you can instruct GL to use them when performing lighting calculations. The three calls to the **lmbind** subroutine in the example program tell the system to use the material, light source, and light model defined by the **lmdef** subroutine.

It is more efficient to define surface materials, light sources, and lighting models before activating them with the **lmbind** subroutine because more computation is involved in defining models than in invoking them. By predefining all the materials, light sources, and lighting models to use in your application, you can quickly switch among different materials and lighting sources when drawing a scene.

There can be only one active light model and surface material at any one time. However, you can use up to the value of MAXLIGHTS lights, where MAXLIGHTS is a constant defined in the **/usr/include/gl/gl.h** file. (The value is 8 for the High Performance 3-D Color Graphics Processor, and should not be changed.) The example program **cylinder2.c** (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) uses multiple lights.

The final new subroutine is the **n3f** subroutine. Use it to define surface normal vectors. Look at the **n3f** subroutine called from the `draw_cylinder` subroutine of the example program (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) . In the simple lighting calculation, the resultant color at a vertex depends on the angular relationship between the surface normal at that vertex and the light source direction.

The call to the **n3f** subroutine sends a normal vector of unit length to the graphics hardware to be transformed and subsequently used for a lighting calculation. Each time a different normal is sent to the

graphics hardware, a lighting calculation is performed and the resultant color is associated with the vertex (using the **v3f** subroutine) following the **n3f** command.

If you have not done so already, try running the example program **cylinder1.c**. Notice as the cylinder rotates, the color changes to different shades of gray. This is because the angular relationships between the surface normals and the light source (which happens to be located behind our eye along the *z* axis) is changing.

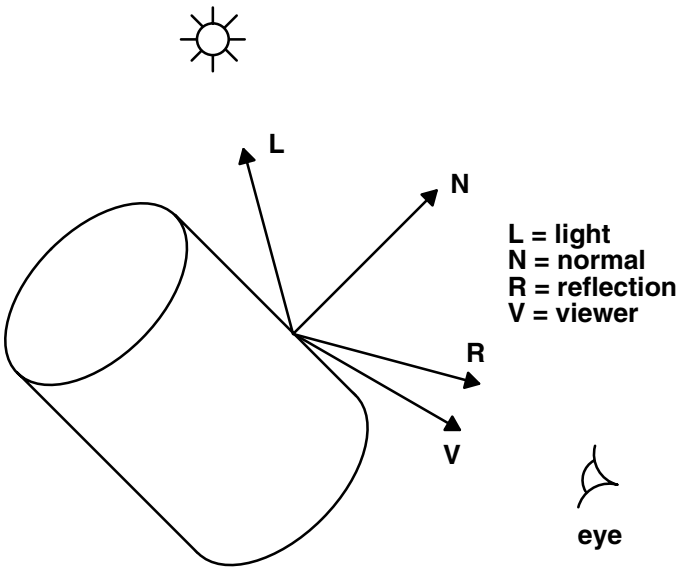
Also, note that as the interior of the cylinder comes into view, the interior color appears a constant shade of dark gray. This is because the surface normals are facing away from the light source, so the interior of the cylinder is illuminated only by ambient light, which is independent of the direction of the light source.

Now that you have seen how to implement a simple lighting calculation, you can expand the capabilities of the lighting calculation to include specularity and multiple surface materials and lights.

Specularity

What if the cylinder in the example were made of a shiny metal instead of a rough material? Compared to rough or diffusely reflecting surfaces that reflect light equally in all directions, smooth surfaces tend to reflect light in a directional manner. This directional reflection, called specular reflection, is what causes highlights.

On a perfectly smooth surface such as a mirror, the direction of a reflected ray is equal to the angle of incidence (that is, the angle between the light direction vector and the surface normal vector). On surfaces that are less than perfectly smooth there will be some scattering of the reflected light. Thus, the angle of the reflected light is weighted towards, but not always equal to, the angle of incidence (see the following ″figure″ on page 112).



Specularity

Calculations for diffusely reflected light from a light source depend on the angular relationship between the surface normal and light source direction, as well as the reflectance characteristics of the surface. In addition to reflectance characteristics, specularly reflected light depends on the angular relationships between the surface normal, light source direction, and view direction. When the view direction coincides with the direction of the reflected light rays, the viewer sees a specular highlight or glare.

The surface material of the cylinder can be made to appear smoother by incorporating specular reflections. You can do this by changing the def_simple_light_calc subroutine to look like this:

```
def_simple_light_model(){
   lmdef(DEFMATERIAL, 1, 11, shiny_material);
   lmdef(DEFLIGHT, 1, 0, NULL);
   lmdef(DEFLMODEL, 1, 0, NULL);
}
```

At the module level of the program, include the definition for the shiny_material property array:

```
float  shiny_material[] = {
         SPECULAR, 0.8, 0.8, 0.8,
         DIFFUSE, 0.4, 0.4, 0.4,
         SHININESS, 30.0,
         LMNULL};
```

Now we have provided a non-NULL properties array; values are available for further lighting calculations. Properties are set by specifying a property identifier followed by the expected values for that property. In our example, we set the specular reflectance (using SPECULAR as the property identifier) of the surface for the red, green, and blue components of white light to [0.8, 0.8, and 0.8], respectively. Likewise, the diffuse reflectance was set to [0.4, 0.4, 0.4].

Reflectance components vary between 0.0 and 1.0 (0.0 being 0% reflective and 1.0 being 100% reflective). The shininess property indicates how smooth or shiny the surface appears. The higher the number, the smoother the surface, and subsequently the more focused the specular highlight.

The values for SHININESS can range from 0.0 (no specular highlight) to 128.0 (very focused specular highlight). The shininess specified should be a whole number. When a property identifier and the corresponding values are specified in a call to the **lmdef** subroutine, the new value for that property overrides a default that is provided by GL. This way, you have to specify only those properties whose default values you do not want.

You must always end a property list with the LMNULL token. This token lets GL know that you are finished specifying properties. There are more properties for surface materials that we have not discussed as well as other properties that apply specifically to lighting models and light sources. A complete description of all the properties available to the **lmdef** subroutine call and their defaults appears in the section on Lighting Subroutines .

If you run the example program with these changes, you will notice how much shinier the cylinder looks. Watch how the highlight appears when the surface normals point at your eye and disappear as they move away.

Multiple Surface Materials and Multiple Lights

The next example program displays two intersecting cylinders, using a different surface material for each cylinder. You can also light each cylinder with two light sources. Study the example program **cylinder2.c** (found in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*) .

At the top of the program, a second property list is defined for a new material called purple_material and a property list for a second light called blue_light. In the blue_light property list, the light direction is specified as [0.0, 1.0, 0.0, 0.0]. The first three numbers specify the *xyz* direction of the light. The direction vector [0.0, 1.0, 0.0] indicates that the light direction is along the *y* axis pointing towards the origin.

In other words, the blue light is above the cylinders and pointing toward them. The fourth number (0.0) indicates that the light is positioned infinitely far away along the direction vector (in our example the *y* axis). Differences between infinite and noninfinite lights are discussed in Advanced Lighting Capabilities.

The LCOLOR specifies the RGB color of the light. This light has only blue color. By varying the values of each of the three color components between 0.0 and 1.0 you can vary the intensity of the light. That is, setting the blue light color to [0.0, 0.0, 0.6] produces a more intense blue than it would have been if you had set the color to [0.0, 0.0, 0.3].

In the def_light_calc subroutine, additional material and light source are defined using the **lmdef** subroutine. In the use_light_calc portion of the program, both light sources are bound because both are used during the entire animation. However, because you want to switch back and forth between surface materials, the program does not bind the surface material until ready to use it. In the main loop, the program performs an

```
lmbind(MATERIAL, 1)
```

before drawing the first cylinder and then another

```
lmbind(MATERIAL, 2)
```

when drawing the second cylinder.

When you run the example program **cylinder2.c**, notice that the first cylinder has the same surface material as the cylinder drawn by the example program **cylinder1.c**, but that the second cylinder consists of a duller, purple material. In addition, each cylinder reflects different amounts of the blue overhead light.

Advanced Lighting Capabilities

Now that you are familiar with the basic lighting capabilities of GL, the following sections discuss some of the advanced lighting features.

- Material Emission
- More on Ambient Light
- lmcolor Subroutine
- Local Viewer
- Local Lights
- Light Attenuation

Material Emission

One property of surface materials is *emissivity*, the amount of light radiated (not reflected) by the material itself. A material can be made self-luminous by adding the identifier EMISSION to the property list followed by the red, green, and blue emission components (each with a value between 0.0 and 1.0). For example, we could define the properties list for a material as the following:

```
float glowing_material[] = {
            EMISSION, 0.8, 0.25, 0.0,
            AMBIENT, 0.0, 0.0, 0.0,
            DIFFUSE, 0.0, 0.0, 0.0,
            SPECULAR, 0.0, 0.0, 0.0,
            LMNULL};
```

The material would appear to emit orange light. The ambient, diffuse, and specular reflectance values are specifically assigned to 0.0 so the color of the object is not affected by light sources. If the coefficients are not zeroed out, GL uses the default nonzero values.

Using a material definition such as glowing_material that has emissive but no reflective properties is useful for simulating lights at night. It is important to distinguish between the way the lighting facility handles surface materials with emissive properties and the way it handles light emitted from light sources. Unlike a light source, emitted light from a material does not affect the color of any other object.

More on Ambient Light

In defining the simple lighting calculation, the default values for ambient light and ambient reflectance provided by GL were used. However, GL allows you to control the amount of ambient light displayed in the scene in three different ways:

- Color
- Additional, specific light source
- Material reflectance values

You can specify the color of ambient light present in the entire scene. Because the color of ambient light is the same at any point in the scene, the scene ambient light color is a property of the lighting model. For example, in defining a lighting calculation with the **lmdef** subroutine using the property list, the lines

```
float simple_light_model[] =
   {AMBIENT, 0.3, 0.3, 0.3, LMNULL};
```

define a lighting model similar to the simple lighting calculation except that this example overrides the default scene ambient light color with the specified values.

In addition to the scene ambient light color specified in the lighting model, a specific light source can contribute to the ambient light in the scene. Consider the following modification to the property array `blue_light`:

```
float blue_light[] =
      {LCOLOR, 0.0, 0.0, 0.6,
      AMBIENT, 0.0, 0.0, 0.2,
      POSITION, 0.0, 1.0, 0.0, 0.0, LMNULL};
```

Now, the ambient light associated with `blue_light` is added to the scene ambient light once `blue_light` has been defined and bound. Unlike the scene ambient light, the ambient contribution from the blue light disappears if you turn the blue light off. It is important to note that ambient light associated with a particular light source is omnidirectional just like ambient light specified with the lighting model.

The third method of controlling displayed ambient color is to alter the ambient light reflectance values of the material. You have seen how to specify this property by using `glowing_material` (you set it to [0.0, 0.0, 0.0]). The ambient color displayed for a surface is determined by adding the ambient contributions from the light model and all of the light sources and multiplying the sum by the material's ambient light reflectance values.

lmcolor Subroutine

You can also use the **lmcolor** subroutine to change the lighting components of an object while the program is running.

The **lmcolor** subroutine lets you change the properties of the currently bound material. It provides a high-performance path to the hardware that would not be otherwise available. Normally, to change the properties of the currently bound material, one would have to redefine the material with the **lmdef** subroutine and then rebind it with the **lmbind** subroutine. The **lmcolor** subroutine helps avoid some of the software overhead involved in redefinitions and rebindings.

The **lmcolor** subroutine works by redirecting the target of the RGB mode color subroutines (**RGBcolor**, **c**, and **cpack**). Normally, these subroutines set the current color. If lighting is off, the color subroutines can be made to affect material properties instead. The syntax is as follows:

```
void lmcolor(Int32 mode)
```

This function accepts the following values for mode:

LMC_COLOR	RGB color commands set the current color. If a color is the last thing sent before drawing a vertex, the vertex is drawn in that color; if a normal is the last thing sent before drawing a vertex, the vertex is lighted (the default mode).
LMC_EMISSION	RGB color commands set the emission color property of the currently bound material.
LMC_AMBIENT	RGB color commands set the ambient color property of the currently bound material.
LMC_DIFFUSE	LRGB color commands set the diffuse color property of the currently bound material.
LMC_SPECULAR	RGB color commands set the specular color property of the currently bound material.
LMC_AD	RGB color commands set the diffuse and ambient color properties of the currently bound material.
LMC_NULL	RGB color commands are ignored.

Calls to the **lmdef** subroutine can change properties of the currently bound material, but because it must modify the data structure of the material, this subroutine executes relatively slowly. The **lmcolor** subroutine provides a fast and efficient way to change properties of the currently bound material as maintained in the graphics hardware without changing the definition of the material. This means, however, that all **lmcolor** changes are lost when you bind a new material.

Use the standard RGB-mode color subroutines **RGBcolor**, **c**, and **cpack** to change material properties selected by the **lmcolor** subroutine. When lighting is not active, RGB-mode color commands change the current drawing color. The **lmcolor** subroutine is significant only while lighting is on.

Local Viewer

As stated in the section on pecularity, the intensity and color of the specular highlight seen depends on the view direction. The lighting calculations performed by the system take into account the view direction, although they can do so in one of two different ways. One of the methods, called *infinite viewer*, involves an approximation, but results in significantly improved performance. The other method, *local viewer*, is considerably slower, but more exact. For most applications, there is very little difference in the visual appearance between the two methods, and therefore the infinite viewer is the default.

The infinite viewer makes the approximation that the eye (as far as the lighting calculations are concerned) is infinitely far away. This does not mean that the transformation matrices are somehow altered; they are not. The geometry of the scene being drawn is completely unaffected; only the colors that come out of the lighting equations are affected.

When this assumption is made, the view direction vector remains constant throughout the scene. Using an infinite viewer is beneficial to application performance because the system does not have to recompute the view direction vector for every vertex in the scene (recomputing this vector is computationally expensive because it requires a square root operation).

However, describing the viewer as located infinitely far away is not as realistic as placing the viewer at some finite position. To define a local viewer, set the local viewer property in the property list for the lighting model to true (1.0):

```
float local_viewer_model[] = {LOCALVIEWER, 1.0, LMNULL};
```

> Note: Because the local view vector changes with each vertex, the lighting computation must be performed when the vertex subroutine (**v3f**) rather than the normal (**n3f**) subroutine is issued. When performing the lighting calculation at the vertex, the graphics hardware uses the normal vector from the most recently issued normal subroutine.

The difference between infinite and local viewer is significant only when the surface being drawn is extremely close to the eye. For normal drawing, the performance improvement of the infinite viewer should significantly outweigh the improved appearance of the local viewer.

Local Lights

Local lights are analogous to a local viewer and are available through the lighting facility. To convert the blue light from the earlier example to a local light, change the property list definition to the following:

```
float blue_light[] = {
   LCOLOR, 0.0, 0.0, 0.6,
   POSITION, 0.0, 5.0, 0.0, 1.0, LMNULL};
```

Changing the fourth positional component from a 0.0 to a 1.0 tells the system that this light is local and that the *x*, *y*, and *z* components specify a light position rather than a light direction. This is a significant difference because the light direction vector becomes the vector from the current vertex to the position of the light. Thus, you must take care not to position the light too close to the surface. If you are not careful, you may find that you positioned the light source within or below a surface in your scene (notice the *y* component of the position is changed from a 1.0 to 5.0 in the property list for `blue_light`). Like a local viewer computation, the lighting computation is performed when a vertex subroutine is issued when you are using a local light.

Using a local viewer or local lights, or both, makes a significant visual difference when you are displaying geometry that contains flat surfaces. The normal vector across a flat surface is constant. Thus, if you are using infinite light sources and an infinite viewer, the angular relationships between the normal, view, and light vectors remain constant, resulting in a constant color across the surface. Using a local viewer or lights, or both, causes the view or light vectors, or both, and the resultant color to change across the flat surface. (See the example program **platelocal.c** (in in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)* .)

Light Attenuation

As you move a light further away from an object, the effect of the light on the object diminishes. This is referred to as light attenuation. If you are using local lights in your lighting model, the lighting facility will attenuate them if desired. The amount of attenuation is a function of the distance from the current vertex to the light source. Because the attenuation function used is the same for all local lights, the attenuation function is a property of the light model rather than of a particular light. The attenuation function used is as follows:

$$\frac{1.0}{K_{off} + K_{rate} \times Dist}$$

where Koff equals the constant scene attenuation factor, fixed; Krate equals the scene attenuation rate; and Dist equals the distance from current vertex to light source.

The constant scene attenuation factor sets the minimum attenuation that a light will undergo. It should be set to a value greater than zero. The attenuation rate controls how fast the attenuation sets in as a light is moved away. It should not be set to a negative value. The resulting attenuation factor computed at a vertex is clipped to [0.0, 1.0] and is multiplied with the light source color in order to attenuate it.

To specify attenuation in the light model property list, specify the constant attenuation parameter followed by the attenuation rate parameter:

```
float local_light_model[] = {AMBIENT,  0.3, 0.3, 0.3,
         LOCALVIEWER, 1.0,
         ATTENUATION, 1.0, 1.5, LMNULL};
```

The fixed scene attenuation factor dampens the overall attenuation function. By increasing the value of the fixed attenuation factor, you lessen the effect of the distance-dependent attenuation. It is a good idea to

start off with the fixed attenuation factor equal to 1.0. That way, when the distance from the vertex to the light is zero, there is no attenuation. Furthermore, if you make the attenuation rate factor large, the illumination falls off too quickly.

There are two reasons why the formula shown is used for attenuation, rather than some different form (for example, an inverse square law):

- One reason is based on physical principles. Although it is true that light from a POINT source falls off as the square of the distance, this is not true when one is near an extended light source.
- The other reason that the inverse linear form is preferred has to do with monitor phosphors and the dynamic range of typical monitors. No monitor can be as bright as the sun, nor as dark as a cave. Their dynamic range does not approach the capabilities of the human eye. An inverse square attenuation (or an exponential attenuation) leads to very large changes in brightness. Such attenuations normally result in a very poor visual appearance on the monitor.

Lighting in Matrix Mode

This section outlines the technique of creating lighting effects in matrix mode:

- Transforming Vectors into Normalized Device Coordinates
- Positioning the Lights

Transforming Vectors into Normalized Device Coordinates

If you are not doing lighting calculations, the geometric transformation pipeline is relatively simple. Vertices (vectors) representing positions in 3-D space are transformed into normalized device coordinates (NDCs, that is, 3-D cubes whose *x*, *y*, and *z* coordinates are restricted to lie between -1.0 and 1.0) and then scaled to the physical window (and screen) integer coordinates.

The transformation to NDCs is accomplished by multiplying the input vector by a matrix that represents the combined actions of modeling, viewing, and projection transformations. Each of the individual transformations is represented by a matrix, and these can all be multiplied together to yield one matrix that has the same effect as the sequential application of all the individual matrices. (See Working with Coordinate Systemsfor more information.)

For lighting calculations, the transformation is done in two steps because the calculations require vectors transformed by the modeling and viewing matrices but not yet projected. The **mmode** subroutine is required to put the system into the two-step mode when lighting is turned on.

In the one-step mode, all transformations are kept on one transformation stack, and all transformation subroutines operate on that stack. In the two-step mode, modeling/viewing transformations are kept separate from the projection transformations. The modeling/viewing transformations are kept on a stack; the projection matrix is kept separately and is not on a stack.

There are three matrix modes: MSINGLE, MVIEWING, and MPROJECTION. The default is MSINGLE, the one-step mode. The system can be placed in the one-step mode by calling

```
mmode(MSINGLE);
```

although, by default, it is already in that mode. To use lighting the system MUST be placed in the MVIEWING mode. This is done by making the call

```
mmode(MVIEWING);
```

After this call, all matrix subroutines operate on the matrix stack. The **getmatrix** subroutine returns the top matrix on the modeling/viewing stack; the **loadmatrix**, **multmatrix**, **rotate**, **translate**, **scale**, **lookat**, and **polarview** all operate on the top matrix of the stack. Note, however, that the perspective projection subroutines (**perspective**, **window**, **ortho2**, or **ortho**) are an exception: they do *not* operate on the modeling/viewing stack, but rather affect the projection matrix directly.

If the application needs to define a customized perspective transformation (one that is not covered by **perspective, window, ortho**, or **ortho2**) while lighting is turned on, the application should go into MPROJECTION mode by calling

```
mmode(MPROJECTION);
```

In this mode all the matrix subroutines operate on the projection matrix. The **getmatrix** subroutine returns the current projection matrix. The **loadmatrix**, **multmatrix**, **rot**, **rotate**, **translate**, **scale**, **lookat**, and **polarview** all operate directly on the projection matrix, as well as the usual projection subroutines (**perspective**, **window**, **ortho2**, and **ortho**). On the other hand, the **pushmatrix** and **popmatrix** calls make no sense in this mode because the stack is not directly accessible. Furthermore, no drawing should be done while in MPROJECTION mode; the system is not configured for drawing when it is in this mode. When you are finished defining the projection matrix, you should go back directly to MVIEWING mode. Please heed the following note of caution:

> Attention: Entering or leaving MSINGLE mode scrambles the contents of the matrix stack and leaves the current projection matrix undefined.

When you write a GL program, you must always remember to initialize the transformations. By default, they are NOT pre-initialized by the system. In the one-step mode (MSINGLE mode), a call to **loadmatrix** or to one of the four projection subroutines (**perspective**, **window**, **ortho2**, or **ortho**) is sufficient. In the two-step mode, both the projection and the modeling/viewing matrices must be initialized. Again, calls to the projection subroutines are sufficient to initialize the projection matrix; the modeling/viewing matrix can be initialized by a call to the **loadmatrix** subroutine while in MVIEWING mode.

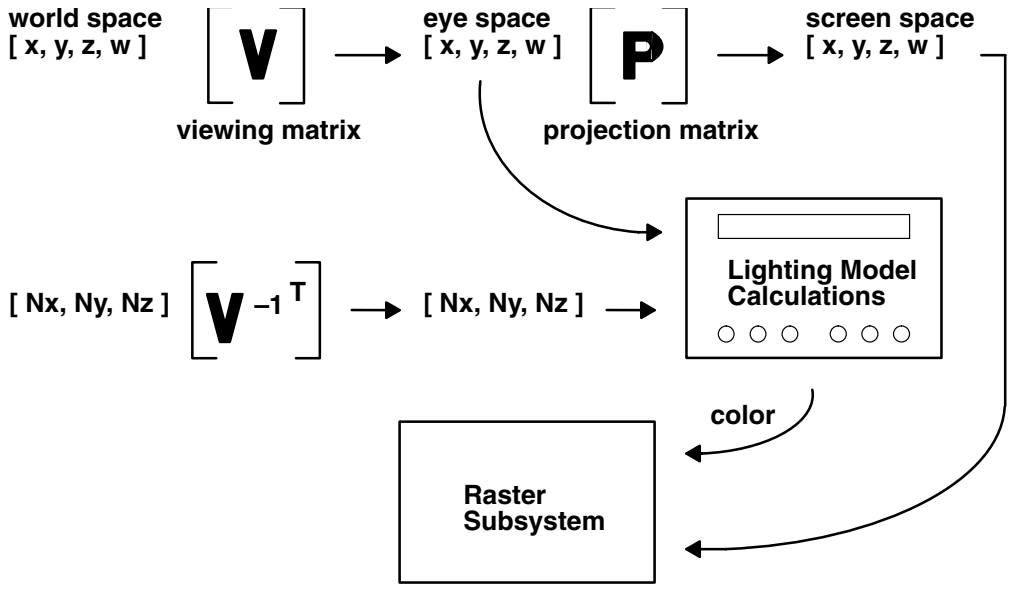
Normally, you load the identity matrix (a 4x4 matrix with ones running along the diagonal, the other entries zero), although any matrix may be loaded. The initialization is required because most of the matrix subroutines (**multmatrix**, **rot**, **rotate**, **translate**, **scale**, **lookat**, and **polarview**) multiply into the current matrix, and if that matrix does not exist, the multiplication cannot take place. Remember, if you intend to use lighting, you must leave MSINGLE mode before performing the initialization.

Like vertices, normal vectors associated with the vertices must also be transformed. However, normals are transformed according to different mathematical rules than vertices. For lighting calculations, the normal vectors are multiplied by the inverse transpose of the 3X3 upper left submatrix of the modeling/viewing transformation.

As you operate on the modeling/viewing stack, its inverse transpose is automatically kept up to date.

The **getmmode** subroutine returns the current matrix mode. The values returned can be compared to the values of MSINGLE, MPROJECTION, and MVIEWING to determine the current mode. Mode identifiers are defined in the **/usr/include/gl/gl.h**file.

The following ″figure″ on page 120 illustrates the calculations done to each vertex sent down the pipeline in MVIEWING mode.



Lighting Calculations

Positioning the Lights

When a light is bound with the **lmbind** subroutine, the location it takes up in world space depends on the transformation on the top of the matrix stack. That is, the coordinates of the light are run through the geometry pipeline just as any other 3-D coordinate would be. This must be kept in mind when rendering complex, animated scenes.

How to draw a scene where the eyepoint changes from frame to frame, while the light remains fixed in world coordinates, or to draw a scene where the light moves about from frame to frame, might not be immediately obvious. Therefore, we summarize the following steps to achieve a moving eyepoint and/or to have moving lights. There are five general cases:

- A static eyepoint and a static light.
- A static eyepoint and a moving light.
- A moving eyepoint and a static light.
- A moving eyepoint and a moving light.
- A moving eyepoint with the light attached to the eyepoint.

The general rules for binding lights are:

- Call `lmbind(LIGHT#, index)` after changing the view.
- Call `lmbind(LIGHT#, index)` only when transformations for the light are premultiplied on the stack.
- Never call `lmbind(LIGHT#, index)` when transformations for surfaces are premultiplied on the stack.

The detailed steps in setting up the viewing transformations, the modeling transformation, and the lighting model specifications for each of these five cases are outlined here. The first four cases all use the following pseudocode:

1. Specify the viewing transformation.
2. Push the transformation stack.
3. Set up the modeling matrix to position the light correctly.
4. Call the **lmbind** subroutine to bind the light source.

5. Pop the transformation stack.
6. Push the transformation stack.
7. Draw the objects.
8. Pop the transformation stack.

A Static Eyepoint and a Static Light

In this scenario, the drawing loop simply repeats steps 6 through 8. Presumably, the objects are drawn in different locations each time through the loop, if you are interested in having moving objects.

A Static Eyepoint and a Moving Light

In this scenario, the drawing loop returns to step 2 and repeats steps 2 through 8. Each time through the loop, change the modeling transformation in step 3 to move the light. You can have either moving or static objects by changing step 7.

A Moving Eyepoint and a Static Light

In this scenario, the drawing loop returns to step 1 and repeats steps 1 through 8. Each time through the loop, change the viewing transformation in step 1 to move the eyepoint. Do not change the modeling matrix in step 3 if you do not want the light to move around in world coordinates. Again, you can have either moving or static objects by changing step 7.

A Moving Eyepoint and a Moving Light

In this scenario, the drawing loop returns to step 1 and repeats steps 1 through 8. Each time through the loop, change the viewing transformation in step 1 to move the eyepoint. Change the modeling matrix in step 3 to move the light around. Again, you can have either moving or static objects by changing step 7.

A Moving Eyepoint with the Light Attached to the Eyepoint

This scenario requires a different sequence of steps:

1. Specify the viewing transformation.
2. Set up the modeling matrix to position the light correctly.
3. Call the **lmbind** subroutine to bind the light source.
4. Specify the viewing transformation again.
5. Push the transformation stack.
6. Draw the objects.
7. Pop the transformation stack.

In this case, the drawing loop returns to step 4 and repeats steps 4 through 7. Each time through the loop, change the viewing transformation in step 4 to move the eyepoint. Again, you can have either moving or static objects by changing step 6.

Lighting Subroutines

The GL lighting facility is implemented with seven additional subroutines to the Graphics Library:

- **n3f** and **normal** subroutines, which set the current normal.
- **mmode** subroutine, which sets the current matrix mode.
- **getmmode** subroutine, which returns the current matrix mode.
- **lmdef** subroutine, which defines a material, light source, or lighting model.
- **lmbind** subroutine, which makes a material, light source, or lighting model current.
- **lmcolor** subroutine, which enables dynamic resetting of the material properties.

All lighting model property names and constants are symbolically defined in the **/usr/include/gl/gl.h** file.

n3f Subroutine

The **n3f** subroutine takes the address of an array of three floating-point numbers (the *vector* parameter) and sets the value for the current vertex normal. Normal vectors are assumed to be of unit length, therefore, the ″equation″ on page 122

$$x^2 + y^2 + z^2$$

should equal 1.0. The normal vector is transformed into eye coordinates using the inverse transpose of the current viewing matrix. It is then stored for use by the lighting equation.

New normals can occur as frequently as every new graphics position and as infrequently as desired. If only one normal is given per polygon and the lighting equation is evaluated on a per-normal basis (when using an infinite viewer and infinite light sources), then the calculations are done only once per polygon. The syntax is as follows:

```
void n3f(Float32 vector[3])
```

normal Subroutine

The **normal** subroutine takes exactly the same parameter as the **n3f** subroutine. The only difference is that **normal** can be used in display lists and **n3f** cannot. The syntax is as follows:

```
void normal(Coord narray[3])
```

mmode Subroutine

The **mmode** subroutine takes one integer parameter (*mode*), which is either MSINGLE, MPROJECTION, or MVIEWING. The different modes are necessary to maintain a separate projection matrix and a separate modeling/viewing matrix and its inverse transpose.

The modeling/viewing matrix transforms model coordinates into eye coordinates where the lighting calculations are performed. The inverse transpose of this matrix is used to transform normals from model coordinates into eye coordinates. The projection matrix transforms eye coordinates into screen coordinates. The figure entitled ″Lighting Calculations″ on page 120 shows how the various matrices operate on display list coordinates and normal vectors to produce screen coordinates and colors.

The lighting equation uses output of the modeling/viewing matrix and its inverse transpose. To use the lighting facility, it is necessary to call `mmode(MVIEWING)` before setting viewing or modeling matrices. This informs the system that future transformation subroutines will affect the modeling/viewing matrix and its inverse transpose. Each of the matrix modes is described fully in the following sections. The syntax is as follows:

```
void mmode(Int16 mode)
```

MSINGLE mode is the default mode. The viewing, modeling, and projection matrices are combined into one matrix, which the **getmatrix** subroutine returns. Because all matrices are combined in MSINGLE mode, there is no way to transform model coordinates into eye coordinates or to transform normals. Therefore, the lighting model facility does not work in MSINGLE mode. Entering or exiting from MSINGLE mode pops the entire matrix stack and leaves the current matrix undefined.

In MPROJECTION mode, all matrix subroutines deal only with the projection matrix. There is only one projection matrix, and the **pushmatrix** and **popmatrix** subroutines result in errors in MPROJECTION

mode. The **getmatrix** subroutine returns the current projection matrix. The modeling/viewing matrix is disabled while in MPROJECTION mode: the results of transforming points in MPROJECTION are undefined.

In MVIEWING mode, all matrix subroutines deal only with the modeling/viewing matrix and its inverse transpose. All matrix subroutines premultiply or load the modeling/viewing matrix by the matrix, and premultiply or load the inverse transpose viewing matrix with the inverse transpose of the matrix.

For the **loadmatrix** and **multmatrix** subroutines, the matrix has to be inverted, so singular matrices will cause an error. In MVIEWING mode, the **getmatrix** subroutine returns the modeling/viewing matrix. The **perspective**, **window**, **ortho**, and **ortho2** subroutines load a projection matrix even in `MVIEWING` mode. Because these subroutines do not affect the modeling/viewing matrix stack and because it is common to build a viewing matrix using subroutines that only multiply matrices, it normally is necessary to load an identity matrix onto the modeling/viewing matrix stack before defining the viewing matrix.

getmmode Subroutine

The **getmmode** subroutine returns the current matrix mode. The values returned can be compared to the values of MSINGLE, MPROJECTION, and MVIEWING to determine the current mode. Mode identifiers are defined in the **/usr/include/gl/gl.h**file. The syntax is as follows:

```
Int32 getmmode()
```

lmdef Subroutine

The **lmdef** subroutine defines a new material, light source, or lighting model and takes four parameters. The first parameter, *deftype*, specifies what is to be defined and is either `DEFMATERIAL`, `DEFLIGHT`, or `DEFLMODEL`. The second parameter, *index*, is the name or index into the table of stored materials, light sources, or lighting models. Indexes for each of these groups are independent. You can define up to 65535 materials, 65535 light sources, and 65535 lighting models. However, index 0 is predefined for each group and cannot be changed.

The third parameter, *numpoints*, is the length of the properties array and is the number of floating point numbers contained within the array. The fourth parameter, *properties*, is the properties array, which is a list of properties to be assigned to the material, light source, or lighting model. Values in the array are property identifiers, each followed by the appropriate number of data values. All identifiers and data values are floating-point numbers. Only property identifiers appropriate for the object being defined should be included in the properties array. The last entry must be `LMNULL (0.0)`. Property identifiers are defined in the **/usr/include/gl/gl.h** file.

All properties have default values that have been chosen for their efficient execution. The first time a material, light, or lighting property is defined, it is initialized to the default values. A definition can be set to all default values by calling the **lmdef** subroutine with either a null pointer to the property array (C programming language only) or with `LMNULL (0.0)` as the first and only property identifier.

Incremental changes can be made to a material, light source, or lighting model definition. Each call to the **lmdef** subroutine changes only the properties included in its properties array. Properties that are not specified in the properties array keep their previous values. Any valid property can be changed regardless of whether that property is relevant to the current lighting calculation. However, changes made to a definition that is currently bound are effective immediately.

The format of the properties array is a sequence of property identifiers each followed by the appropriate number of data values. The last array entry must be LMNULL. Described below are the material, light source, and lighting model properties along with the number of data values that follow each identifier. Each property is called by its symbolic name. The syntax is as follows:

```
void lmdef(Int16 deftype, Int32 index,
           Int16 numpoints, Float32 properties[])
```

Material Properties

Material properties are all the properties a material groups together to define its surface characteristics, such as diffuse reflectance and shininess. The available material properties are as follows:

EMISSION	The EMISSION property is the emission color of the material. Following the property identifier should be the new R, G, B color values where 0.0 <= R, G, B <= 1.0.
AMBIENT	The AMBIENT property is the ambient reflectance of the material. Following the property identifier should be the new R, G, B color values where 0.0 <= R, G, B <= 1.0.
DIFFUSE	The DIFFUSE property is the diffuse reflectance of the material. Following the property identifier should be the new R, G, B color values where 0.0 <= R, G, B <= 1.0.
SPECULAR	The SPECULAR property is the specular reflectance of the material. Following the property identifier should be the new R, G, B color values where 0.0 <= R, G, B <= 1.0.
SHININESS	The SHININESS property is the specular light scattering exponent. Following the property identifier should be the new value for the shininess. The values for shininess can range from 0.0 to 128.0 and should represent whole numbers. If shininess <= 0.0 then there will be no specular highlight. This is functionally equivalent to setting the material specular reflectance to [0.0, 0.0, 0.0].
ALPHA	The ALPHA property is the transparency of the material and can be used when performing alpha blending. Following the property identifier should be the new alpha value where 0.0 <= alpha <= 1.0. Certain alpha blending operations require that alpha bitplanes be installed in the system.
COLORINDEXES	Specifies the material properties used when lighting in color map mode.

The default values for material properties are:

EMISSION	0.0, 0.0, 0.0
AMBIENT	0.2, 0.2, 0.2
DIFFUSE	0.8, 0.8, 0.8
SPECULAR	0.0, 0.0, 0.0
SHININESS	0.0
ALPHA	1.0
COLORINDEXES	0.0, 127.5, 255.0

Light Source Properties

Light source properties are all the properties a light source groups together to define its characteristics, such as color and position. The available light properties are as follows:

AMBIENT	The AMBIENT property is the color of the ambient light associated with the light source. Following the property identifier should be the new R, G, B color values where 0.0 <= R, G, B <= 1.0.
LCOLOR	The LCOLOR property is the color of the light source. Following the property identifier should be the new R, G, B color values where 0.0 <= R, G, B <= 1.0.
POSITION	The POSITION property is the position of the light source. Following the property identifier should be the new *x*, *y*, *z*, and *w* coordinates of the light. If the *w* coordinate of the light source is 0.0, the light is an infinite light source and its position specifies the light direction. (Such a light source can be called a directional light source.) The light direction is computed by normalizing the light's position vector. If the *w* coordinate of the light is not 0.0, then the light is a local light and *x*, *y*, *z*, and *w* are divided by *w* to specify the position. Light positions or directions are defined in the current model coordinate space and are affected by the current transformation matrix at the time they are bound.
SPOTDIRECTION	Assigns the direction (axis) in which a spotlight source emits.
SPOTLIGHT	Assigns the spread angle and concentration exponent of a spotlight.

The default values for light source properties are:

AMBIENT	0.0, 0.0, 0.0

LCOLOR	1.0, 1.0, 1.0
POSITION	0.0, 0.0, 1.0, 0.0
SPOTDIRECTION	0.0, 0.0, -1.0
SPOTLIGHT	0.0, 180.0

Lighting Model Properties

Lighting model properties are all the properties of a lighting model, such as scene ambient light and distance attenuation factors. The available lighting model properties are as follows:

AMBIENT
: The AMBIENT property is the color of ambient light in the scene. Following the property identifier should be the new R, G, B color values where 0.0 <= R, G, B <= 1.0.

LOCALVIEWER
: The LOCALVIEWER property informs the system whether the viewer (eye position) is local to the scene. Following the property identifier should be either 1.0 or 0.0 (true or false). If the viewer is local, then the eye position is assumed to be located at (0,0,0) in eye coordinates. When the viewer is local, the vector from the vertex to the eye must be calculated for each vertex. If the viewer is not local, then the viewer is at infinity along the positive *z* axis. The view direction vector is [0,0,1] for all vertices.

  Note: The LOCALVIEWER property affects only the manner in which the lighting calculations are performed. The setting of the LOCALVIEWER does not in any way affect the transformation stack or alter the geometrical position of the surfaces being drawn. It only affects their apparent color (the output of the lighting calculations).

ATTENUATION
: The ATTENUATION property is the scene attenuation factors. Following the property identifier should be the new values for the fixed scene attenuation factor and the variable scene attenuation factor. If either factor is less than 0.0 then it is set to 0.0. If the variable attenuation factor equals 0.0, then lighting attenuation is turned off.

The default values for lighting models are:

AMBIENT	0.2, 0.2, 0.2
LOCALVIEWER	0.0
ATTENUATION	1.0, 0.0

lmbind Subroutine

The **lmbind** subroutine takes two integer parameters: the first, *target*, specifies the target of the bind and the second, *index*, is the index of the source. When a source is bound to a target, it becomes current and subsequent evaluations of the lighting equation use its values. The syntax is as follows:

```
void lmbind(Int16 target, Int32 index)
```

The three types of targets are as follows:

MATERIAL
: If the target of a bind is MATERIAL, the source material becomes the currently bound material. There is only one material target and therefore only one currently bound material.

  Source materials are specified using the same index as when the material was defined using the **lmdef** subroutine. For example,

  ```
  lmbind (MATERIAL, 2)
  ```

  binds material definition 2 to the currently bound material. Material 0 is the default material and disables lighting calculations. This is the most efficient method to disable the lighting calculations. It is functionally equivalent to binding lighting model 0. If an undefined material is the source for the **lmbind** subroutine, material 0 is bound instead.

LIGHTS
: There are MAXLIGHTS lights available as targets (LIGHT0, LIGHT1, and so on), and therefore a MAXLIGHTS number of current lights. If the target of a bind is a light, then the source light replaces whatever light was previously bound to the target. The replaced light is turned off and the newly bound light is turned on. To turn a light off, bind light index 0 to the target light.

  Source lights are specified using the same index as when the light was defined using the **lmdef** subroutine. For example,

  ```
  lmbind (LIGHT3, 54)
  ```

  binds user light definition 54 to system light 3. Light index 0 is the default light and while bound disables lighting calculations for the system light target. If an undefined light is the source to the **lmbind** subroutine, light 0 is bound instead.

  When a local light is bound, its position is transformed by the current modeling/viewing matrix and stored. If the light is infinite, its position is taken as its direction and is also transformed by the current modeling/viewing matrix and stored. Thus, by binding a light after some modeling transformations, a light can easily be made part of an object that is moved through the scene (for example, a handheld candle).

LMODEL
: If the target of a bind is LMODEL, then the source lighting model becomes the current lighting model. There is only one lighting model target and therefore only one current lighting model.

  Source lighting models are specified using the same index as when the lighting model was defined using the **lmdef** subroutine. For example,

  ```
  lmbind (LMODEL, 1)
  ```

  binds lighting model 1 to the current lighting model. Lighting model 0 is the default lighting model and disables all lighting calculations. If an undefined lighting model is the source for the **lmbind** subroutine, lighting model 0 is bound instead.

lmcolor Subroutine

The discussion of the **lmcolor** subroutine appears in Advanced Lighting Capabilities.

Lighting Execution Time and Performance

The lighting equation is evaluated whenever the normal or the graphics position changes, depending on whether the viewer and light sources are local. If the viewer is local, or any of the light sources are local, then the lighting calculation is performed when the graphics position changes (for example, when a **v3f** command is issued).

When the viewer is local, the vector from the vertex to the eye is different for each vertex and has to be calculated for each point. If the viewer is at infinity, then view vector is constant for all vertices and the calculation time is less. When a light source is local, the vector from the vertex to the light source is different for each vertex and has to be calculated for each vertex. If a light is at infinity, then light direction vector is constant for all vertices and the calculation time is less.

Performance is highest when an infinite viewer and a single infinite light source are used in the lighting calculation. Execution time increases slightly for each infinite light source added to the computation. However, the addition of local light sources increases the execution time noticeably.

After a normal is transformed by the inverse transpose of the modeling/viewing matrix, it must be renormalized if the transforming matrix is not orthonormal. This renormalization takes additional time and results in lower performance.

A matrix is orthonormal if each of its row vectors is of unit length and orthogonal to the others. Rotations and translations are always orthonormal. Shear transformations are never orthonormal. Because GL does

not have any built-in shear transformations, the only transformations that might generate a nonorthonormal modeling/viewing matrix are the **multmatrix** and **loadmatrix** subroutines, and the **scale** subroutine when the *x*, *y*, and *z* scale factors are not equal.

> **Note:** If the transformation has been made nonorthonormal by introducing a shear or other nonorthonormal transformation, the system is automatically renormalizing normal vectors. In such a case, you do not need to feed normalized normal vectors to the system because they are renormalized anyway. Note, however, this should not be used as a trick to avoid explicit normalization by the user application.

Formula for Lighting Calculation

Lighting is calculated by the use of the following formula:

$$I^v = I^{lmamb}k^{amb} + \sum_{i=0}^{num\_lights} k^{emis}(\hat{n} \cdot \hat{l}_i) + l_i^{amb}\ k^{amb} + p_i l_i^{inc}\ k^{diff}\ \hat{n} \cdot \hat{l}_i$$

$$+ p_i\ l_i^{inc}k^{spec}(\hat{r}_i \cdot \hat{v})^s$$

The symbols are as follows:

I^v is the color to be assigned to the vertex.

I^{lmamb} is the pervading ambient light (set with AMBIENT in LMODEL).

l_i^{amb} is the ambient intensity of the i^{th} light (set with AMBIENT in LIGHT).

l_i^{inc} is the direct intensity of the i^{th} light (set with COLOR in LIGHT).

p_i is the amount of light i is attenuated (computed internally if ATTENUATION has been set in LMODEL).

k^{emis} is the surface emissivity (set with EMISSIVITY in MATERIAL).

k^{amb} is the ambient reflectivity (set with AMBIENT in MATERIAL).

k^{diff} is the diffuse reflectivity (set with DIFFUSE in MATERIAL).

k^{spec} is the specular reflectivity (set with SPECULAR in MATERIAL).

s is the shininess of the specular reflection (set with SHININESS in MATERIAL).

n is the unit normal vector to the surface (set with either the **normal** or **n3f** subroutine).

l_i is a unit vector pointing to the i^{th} light source (computed internally).

r_i is a unit vector, the reflection of l_i about the normal (computed internally).

v is a unit vector, pointing toward the viewer (computed internally).

How r, l, v and p Are Computed

The unit view vector (*v*) is given by the following equation:

$$\hat{v} = \frac{\vec{x}}{|\vec{x}|} \qquad \text{if LOCALVIEWER is on (true)}$$

$$\hat{v} = -\hat{z} \qquad \text{if LOCALVIEWER is off (false)}$$

The unit light vector (*l*) is given by the following equation:

$$\hat{l}_i = \frac{\vec{L_i} - \vec{x}}{|\vec{L_i} - \vec{x}|} \qquad \text{if light is local (w ! = 0)}$$

$$\hat{l}_i = \frac{\vec{L_i}}{|\vec{L_i}|} \qquad \text{if light is at infinity (w = 0)}$$

The reflection vector (*r*) is given by the following equation:

$$\hat{r}_i = (2\, \hat{l}_i \cdot \hat{n})\, \hat{n} - \hat{l}_i$$

The attenuation factor (*p*) is given by the following equation:

$$p_i = \frac{1}{K_{off} + K_{rate} | \vec{L_i} - \vec{x} |}$$

The attenuation factor is then clipped to a value from 0.0 to 1.0.

The symbols are as follows:

$\vec{x}$ is the three-space position (x/w, y/w, z/w) of the vertex.

$\vec{L_i}$ is the location of the i^{th} light in three-space set (set with POSITION in LIGHT).

$\hat{z}$ is a unit vector pointing in the positive (negative) z direction (z = (0,0,1)).

K_{off} and K_{rate} are the attenuation offset and rate constants (set with ATTENUATION in LMODEL).

Chapter 8. Performing Depth-Cueing

Depth-cueing makes an image appear 3-D by drawing those points brighter that are nearer to the viewer. In depth-cue mode, the intensities of all the polygons, lines, and points drawn to the screen vary according to their *z* values. Depth-cue mode is invoked by the **depthcue** subroutine. For more information on performing depth-cueing, see:

- List of GL Depth-Cueing Subroutines
- Depth-Cueing in Color Map Mode
- Depth-Cueing in RGB Mode

List of GL Depth-Cueing Subroutines

depthcue	Turns depth-cueing on and off.
getdcm	Indicates whether depth-cue mode is on or off.
lRGBrange	Sets the range of color indexes to use for depth-cueing in RGB mode.
lshaderange	Sets the range of color indexes to use for depth-cueing in color map mode.

Depth-Cueing in Color Map Mode

For depth-cueing to work properly, the color map locations specified by the **lshaderange** subroutine must be loaded with a series of colors that gradually increase or decrease intensity. The **lshaderange** subroutine specifies the low-intensity color map index (lowindex) and the high-intensity color map index (highindex). These values are mapped to the low and high *z* values specified by *z*1 and *z*2. The high index must be greater than the low index and the difference between the high index and the low index must be less than the difference between *z*1 and *z*2.

The values of *z*1 and *z*2 should correspond to or lie within the range of *z* values specified by the **lsetdepth** subroutine, which delineates the entire transformation range. The **lshaderange** subroutine specifies the range of values where all of the shading is to occur.

The entries for the color map between the low index and the high index should reflect the appropriate sequence of intensities for the color being drawn. When a depth-cued point is drawn, its *z* value is used to determine its intensity. When a depth-cued line is drawn, the intensities of its points are linearly interpolated from the intensities of its endpoints, which are determined from their *z* values. You can achieve higher resolution if the near and far clipping planes bound the object as closely as possible.

Depth-Cueing in RGB Mode

Depth-cueing in RGB mode works in much the same way as in color map mode. The only difference is that a color ramp does not need to be loaded in the color map. Instead, the near and far RGB colors are specified, and intermediate points are colored by linearly interpolating between the near and far colors.

In general, the color map mode version of depth-cueing is more interesting and versatile, because one has direct control over the intermediate colors. For instance, every tenth color map entry can be made brilliant white, or deep black, to superimpose a depth grid on the figure.

Chapter 9. Configuring the Frame Buffer

Information on configuring the frame buffer for GL includes the following topics:

- List of GL Frame Buffer Configuration Subroutines
- Understanding the Frame Buffer
- Working in Color Map and RGB Modes
- Creating Animated Scenes
- Underlay and Overlay Modes
- Alpha Blending Modes
- Writemasks and Logical Operations
- Clearing, Resetting, and Initializing GL

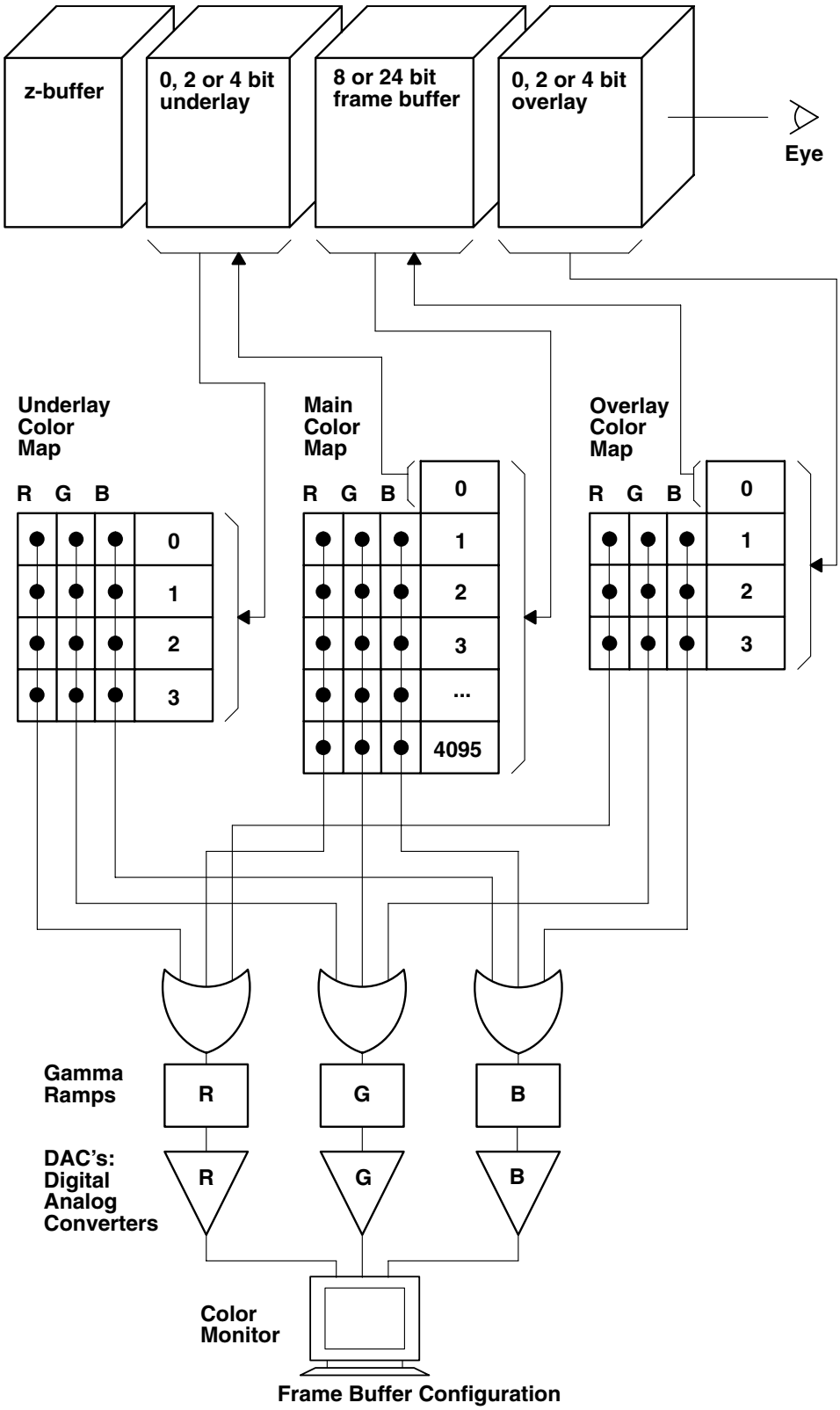
The main color frame buffer stores GL image data pixel by pixel and makes the data available for display. The memory in the main frame buffer can be addressed in single or double buffer mode. You can set the number of bitplanes available for overlay and underlay. Z-buffering is available for three-dimensional calculations, and query functions enable you to determine the current configuration of the frame buffer.

List of GL Frame Buffer Configuration Subroutines

backbuffer	Enables drawing in the back buffer.
doublebuffer	Sets the display mode to double buffer mode.
drawmode	Chooses a set of bitplanes for drawing.
frontbuffer	Enables drawing in the front buffer
getdrawmode	Returns the current drawing mode.
getgdesc	Returns information about currently installed graphics hardware.
getplanes	Returns the number of available bitplanes.
getzbuffer	Indicates whether z-buffering is on or off.
overlay	Sets the number of bitplanes used for overlay.
singlebuffer	Sets the display mode to single buffer mode.
swapbuffers	Exchanges the front and back buffers.
underlay	Sets the number of bitplanes used for underlay.
zbuffer	Enables or disables the z-buffer for storing depth information.
zdraw	Enables drawing to the z-buffer.

Understanding the Frame Buffer

Effective use of the full capabilities of GL requires an understanding of the frame buffer and its organization. GL supports overlay planes, underlay planes, a double-buffered “main” frame buffer, a z-buffer, an alpha buffer, color maps, and gamma ramps (see the figure entitled ″Frame Buffer Configuration″ on page 132). These can be turned on or off, reconfigured, masked with writemasks, and so on. However, the actual number of bitplanes in each buffer varies from adapter to adapter, and therefore the ways in which these bitplanes can be configured and controlled vary from adapter to adapter. The following sections present some basic concepts about the operation of a frame buffer. Following these brief discussions are descriptions of the characteristics of the different systems.


z-buffer
0, 2 or 4 bit underlay
8 or 24 bit frame buffer
0, 2 or 4 bit overlay
Eye
Underlay Color Map
Main Color Map
Overlay Color Map
R G B
0
1
2
3
...
4095
Gamma Ramps
R
G
B
DAC's: Digital Analog Converters
Color Monitor
Frame Buffer Configuration

Main Color Buffer

The main color buffer can be thought of as having the width and height of the window and a depth that can vary from 8 bits to 48 bits, depending on the current configuration and the installed adapter. Pixels stored in this main buffer can be interpreted through a downstream color map (that is, the values stored in the frame buffer are color map indexes), or they can be interpreted directly as red, green, blue (RGB) values, with no intervening color maps. (For more information on how to use or bypass the color maps, see Working in Color Map and RGB Modes.)

The main color buffer can be single buffered or double buffered, depending on the current configuration and the installed adapter. In double buffer mode, the main color buffer is divided in two: the front buffer, which is visible, and the back buffer, which is invisible but can be drawn into. Double buffering is an important technique used to provide smooth animation of moving pictures. (For more information on how to use double buffering, see Creating Animated Scenes.)

The color map configuration and single and double buffer configuration can be changed dynamically by making appropriate GL subroutine calls. The scope of such reconfiguration extends only over the current drawing window.

Multiple windows can appear on the screen, each independently double buffered, each in color map or RGB mode independent of the others. There is no limit to the number of such windows.

Overlay and Underlay Buffers

Like the main color buffer, the overlay and underlay buffers have the width and height of the window. Their depth can range from 0 to 4 bits deep, depending on the installed adapter and the current configuration. Overlay and underlay bitplanes cannot be double buffered and are always in color map mode.

Overlays and underlays function exactly as the names suggest: drawing into the overlay buffer obscures the main color buffer, and drawing into the main color buffer obscures the underlay. (For more information on overlays and underlays, consult Underlay and Overlay Modes.)

Some sophisticated applications find that four overlay and four underlay planes are not enough. Multiple overlays or underlays, even double-buffered overlays and underlays, can be created by appropriately manipulating the color map and the writemasks associated with the main color buffer. Such tricks can easily get quite complicated, and such extra overlays have effectively been stolen from the main color buffer. (For more information on writemasks, see Writemasks and Logical Operations.)

Alpha Buffer

In order for GL to support an alpha blending (transparency) buffer, the installed adapter must contain an alpha buffer. (For more information on alpha blending, see Alpha Blending Modes .)

Z-Buffer

The z-buffer stores the *z* value for each pixel on the screen. The *z* value represents the *z* coordinate, or distance to the eye, for each pixel. When z-buffering is enabled with the **zbuffer** subroutine, the system compares the *z* values for each pixel of any new polygon, line, point, or character to the current *z* value for each pixel and renders only those values representing a distance closer to the eye. (For further discussion of z-buffering, see Removing Hidden Surfaces.)

Query Functions

The query functions available for determining the current frame buffer configuration are the **getdrawmode**, **getplanes**, and **getzbuffer** subroutines.

Working in Color Map and RGB Modes

This section contains discussions on the following topics:

- Color Display, including RGB mode and color map mode
- Onemap and Multimap Mode
- Gamma Correction

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

List of GL Color Map and RGB Mode Subroutines

cmode	Sets color map mode as the current mode.
gammaramp	Defines a color map ramp for gamma correction.
getcmmode	Returns the organization of the current color map.
getmap	Returns the number of the current color map.
getmcolor	Gets a copy of the RGB values for a color map entry.
getmcolors	Returns a range of color map RGB values.
mapcolor	Changes a color map entry to a specified RGB value.
mapcolors	Loads a range of color map entries.
multimap	Organizes the color map as 16 small maps.
onemap	Organizes the color map as one large map.
RGBmode	Sets a display mode that bypasses the color map.
setmap	Selects one of 16 small color maps.

Color Display

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

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

Consider just the red color gun. If no electrons are fired at a pixel, its phosphors do not glow at all, and it appears black. If the gun is turned on to its highest intensity, the phosphor glows bright red. At intermediate intensities, the colors vary between 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 for a pixel is the combination of all three colors. Different combinations of intensity settings of the guns cause a wide variety of colors to appear.

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.

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

RGB Mode

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

Hardware for RGB mode can be built easily if there are 24 bitplanes available. Each one of the three groups of eight bitplanes can be used to store one of the RGB components. Each component is then fed via the digital-to-analog converters (DAC) to the electron guns of the monitor.

Through a technique called dithering, RGB mode is also available on adapters with fewer than 24 bitplanes; in particular, the 8-bitplane systems. Because the 256 colors available on a 8-bitplane system are not enough to create full-color images if employed in the most straightforward fashion, another technique, such as dithering, is required to achieve full color realism.

Dithering is similar to the process used by magazines to print full-color photographs with a base of only four colors. To replicate a color that is not in the palette of 256, alternating pixels in the frame buffer are set to either one of the two or more nearest available colors. This creates a faint and very fine checkerboard pattern that, when viewed from a distance, appears to be the desired color. The success of this technique depends on the fact that the pixels are imperceptible at normal viewing distances.

Color Map Mode

Another way to write and interpret the data in the bitplanes is color map mode. Many applications are better suited to color map mode than to RGB mode. Many of the principles of color maps are used in the overlay, underlay, and pop-up menu modes.

Color map mode is useful for representing scalar false color data, such as temperatures, pressures, viscosities, strains, elevations, and material compositions. It also provides a convenient way to achieve blinking or other simple animation. This technique, called color map animation, makes predrawn images appear to move by simply changing the color map.

Color map mode provides a level of indirection between the values stored in the bitplanes and the RGB values displayed on the screen. The value stored in the standard bitplanes (up to 12 bits) is interpreted 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 specify a color for a pixel on the screen, set the corresponding bit in each of the 12 bitplanes to represent a number between 0 and 4095. This number specifies an index into the color map that indicates the red, green, and blue value for that pixel.

Because the system is in color map mode by default, the lowest 8 values in the color map are loaded as follows:

The Lowest 8 Color Map Values			
Red	**Green**	**Blue**	**Color**
0	0	0	BLACK
255	0	0	RED
0	255	0	GREEN
255	255	0	YELLOW
0	0	255	BLUE
255	0	255	MAGENTA

0	255	255	CYAN
255	255	255	WHITE

In the default color map mode, therefore, an index value of 0 for a given pixel causes that pixel to be displayed as black.

Virtualization of Colormaps

The X server treats colormaps as a shared resource, that is, colormaps are *virtualized* by the X server. Every X client can, but does not have to, request a colormap that is unique to itself and is different from those of the other clients. Under certain circumstances (usually when the client has “focus”), the colormap for that client is loaded into the physical hardware colormap. In such a case, the colors for the given window for the given client appear correct; the colors of the other windows (possibly including the root window) may appear incorrect, if the colormaps of the other windows are different than the currently loaded colormap. The X server does not set the policy for when a given colormap is loaded; this is done by the window manager. Additional information about colormaps and colormap focus policies can be found in the X and Motif documentation.

> Note: Some graphics adapters have more physical hardware colormaps than others. In particular, the 24-bit POWERgraphics GTO, POWER Gt4 and POWER Gt4x adapters have five hardware colormaps each. The X server, in concert with the window manager, is able to manage all five colormaps. If all the clients running on the screen request more than five different colormaps, at least some of the clients will have the incorrect colormap installed. This affects the management of colormaps.

GL Colormaps

The GL paradigm for colormaps is that of a single, hardware colormap. Historically, the GL applications programming interface (API) did not virtualize colormaps; it simply provided access to a single hardware colormap. In the integration of GL with X windows, this behavior is emulated with special code. Thus, the **mapcolor** and **mapcolors** subroutines set a single colormap kept within the X server. A change to this colormap by one GL application affects not only all windows of that application, but all GL applications. Thus, GL applications must cooperate in the sharing of this colormap to obtain high quality visual results.

Using X11 Colormaps with GL

GL supports the use of X11 colormap management subroutines to set and change the colormap(s) associated with a GL window. If the X11 colormap routines are not used, GL creates a single instance of a colormap and associates it with each GL window that is created. This single colormap is accessed with the **mapcolor** and **mapcolors** subroutines. Use of this default colormap can be side-stepped by using the X Window System to create a separate colormap, and associate it to a GL window.

If X11 is used to manage the colormaps for GL windows, then X11 must also be used to allocate the entries in the colormap. The **mapcolor** and **mapcolors** subroutines can be used to change the colors only in the default, private GL map and not in any separately created colormaps.

An example showing how X11 colormap facilities can be used with GL is found in **/usr/lpp/GL/examples/Xcolormap.c.**

> Note: On the POWERgraphics GXT1000, be sure that the correct X11 window ID has been obtained. Note the GXT1000-specific information in the **getXdpy** subroutine.

Onemap and Multimap Modes

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. In onemap mode, the lowest order 12 bits in the frame buffer are used to index into a 4096-entry color map.

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

Call the **gconfig** subroutine to activate the onemap or multimap settings.

Subroutines you use with the **onemap** subroutine and **multimap** subroutine are: the **getcmmode** subroutine, which returns the organization of the current color map; the **setmap** subroutine, which selects which of the small maps the system uses in multimap mode; the **getmap** subroutine, which returns the number of the current color map; and the **cyclemap** subroutine, which cycles through the color maps at a selected rate.

multimap Subroutine

The **multimap** subroutine organizes the color map as 16 small maps, each with a maximum of 256 RGB entries. The **multimap** subroutine does not take effect until the **gconfig** subroutine is called. The syntax is as follows:

```
void multimap()
```

onemap Subroutine

The **onemap** subroutine organizes the color map as a single map with 4096 entries. You must call the **gconfig** subroutine for the **onemap** subroutine to take effect. Onemap is the default mode. The syntax is as follows:

```
void onemap()
```

getcmmode Subroutine

The **getcmmode** subroutine returns the organization of the current color map. A return value of FALSE indicates multimap mode, and TRUE indicates onemap mode. The syntax is as follows:

```
Int32 getcmmode()
```

setmap Subroutine

The **setmap** subroutine selects which of the small maps (0 through 15) the system uses in multimap mode. This selection is ignored in onemap mode. The syntax is as follows:

```
void setmap(Int16 mapnum)
```

getmap Subroutine

The **getmap** subroutine returns the number (from 0 to 15) of the current color map. In onemap mode, this subroutine always returns 0. The syntax is as follows:

```
Int32 getmap()
```

cyclemap Subroutine

The **cyclemap** subroutine cycles through color maps at a specified rate. It defines a duration (in vertical retraces, the current map, and the map that follows when the duration lapses. For example, the following routines set up multimap mode and cycle between two maps, leaving map 1 on for 10 vertical retraces and map 3 on for 5 retraces:

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

When you kill a window or attach to a new window, the maps stop cycling. The syntax is as follows:

```
void cyclemap(Int16 duration, Int16 map, Int16 nextmap)
```

Gamma Correction

The light output of any video display is controlled by the input voltage to the monitor. The relationship between input voltage and the brightness of the display, however, is not linear. For instance, assume that 100% of a monitor's input voltage produces 100% brightness. If you reduce the voltage to 50% of its initial value, the monitor might display only 19% of its initial brightness.

To achieve a linear response from the monitor, the system must vary the input voltage by some exponent. The exponent is called the monitor's gamma. Linear response is achieved on standard monitors with a gamma of 2.4. The system uses a hardware lookup table to compensate for nonlinear response.

> Note: Some graphics adapters have more physical hardware colormaps than others. In particular, the 24-bit POWERgraphics GTO, POWER Gt4 and POWER Gt4x adapters have five hardware colormaps each. The X server, in concert with the window manager, is able to manage all five. Note that if all the clients running on the screen request more than five different colormaps, at least some of the clients will have the incorrect colormap installed. This affects the management of gamma ramps.

GL Gamma Ramps

The historical GL paradigm for gamma ramps has been that of a single, hardware gamma ramp that affected the entire screen. The **gammaramp** subroutine can be used to load that lookup table. This is the case on the 3D Color Graphics Processor adapter; the X server is ignorant of this particular lookup table and does not manage it.

With the newer adapters, the POWERgraphics GTO, the POWER Gt4 and the POWER Gt4x, this paradigm has been modified. When in RGB mode, the gamma ramp for any particular window can be set independently of any other window. Note that the gamma ramps for these adapters are realized as X colormaps. Thus, the X server engages in the management of the RGB mode gamma ramps, and all of the usual rules and behaviors with regard to X colormap management apply. Note, in particular, that a GL RGB window may appear incorrect if the X server hasn't installed a linear ramp. Note also that if more than five different gamma ramps are created, not all can be installed simultaneously, and some RGB windows may appear with incorrect colors.

gammaramp Subroutine

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

The **gammaramp** subroutine affects the entire screen and all running processes. It stays in effect until another call to the same subroutine is made or until the graphics hardware is reset. The default setting has `r[i]=g[i]=b[i]=i` (no modification).

When objects are drawn on the screen in RGB mode, red, green, and blue are stored in the bitplanes, displayed as (*red*, *green*, *blue*), and are the arrays last specified by the **gammaramp** subroutine. Similarly, in color map mode if color `i` is mapped to red, green, blue, objects written in color *i* are displayed as (*red*, *green*, *blue*). The syntax is as follows:

```
void gammaramp(Int16 red[256], Int16 green[256], Int16 blue[256])
```

You may want to read how the different color modes are used in Creating Animated Scenes, Creating Lighting Effects, Performing Depth-Cueing, and Removing Hidden Surfaces.

Creating Animated Scenes

This section outlines the techniques for creating animated scenes within the following topics:

- Double and Single Buffering
- Animation Subroutines

List of GL Animation Subroutines

backbuffer	Enables drawing in the back buffer.
blink	Changes the color map entry at a selectable rate.

cyclemap	Cycles between color maps at a specified rate.
doublebuffer	Sets the display mode to double buffer mode.
frontbuffer	Enables drawing in the front buffer.
getbuffer	Finds out which buffers are enabled for drawing.
getdisplaymode	Returns the current display mode.
gsync	Waits for the next vertical retrace period.
singlebuffer	Sets the display mode to single buffer mode.
swapbuffers	Exchanges the front and back buffers.
swapinterval	Defines the minimum time between buffer swaps.

Double and Single Buffering

Animated objects on the screen are created by using a technique called double buffering.

For smooth motion, the system displays a completely drawn image for a certain time (for instance, a few 60ths of a second), then presents the next frame, also completely drawn, during the next time period, and so on.

Double buffering provides this capability. The system's standard bitplanes are divided into two halves, only one of which is displayed. Drawing is typically done into the other, invisible half. When the drawing is complete, the buffers are swapped. The previously invisible buffer (now containing the next frame) becomes visible, and the previously visible buffer becomes invisible and available for drawing the following frame.

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

In double buffer mode, your program addresses frame buffer memory as if it were two buffers, only one of which is available for drawing or for display at a time.

In single buffer mode, a program addresses frame buffer memory as a single buffer whose pixels are always visible. 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.

Animation Subroutines

The following subroutines enable you to create animated scenes.

doublebuffer Subroutine

The **doublebuffer** subroutine sets the display mode to double buffer mode. It does not take effect until you call the **gconfig** subroutine. In double buffer mode, the bitplanes are partitioned into two groups, the front bitplanes and the back bitplanes. Double buffer mode displays only the front bitplanes. Drawing routines normally update only the back bitplanes; the **frontbuffer** and **backbuffer** subroutines can override the default. The **gconfig** subroutine sets the value for the **frontbuffer** subroutine to False and the **backbuffer** subroutine to True in double buffer mode. The syntax follows:

```
void doublebuffer()
```

swapbuffers Subroutine

No matter what you are doing, the display hardware in the system constantly reads the contents of the visible buffer (the front buffer in double buffer mode), and displays those results on the screen. On a standard monitor, the electron guns sweep from the top of the screen to the bottom, refreshing all pixels,

60 times each second. If the graphics hardware changes the contents of the visible frame buffer, the next time the refresh hardware reads a changed pixel, the new value is drawn instead of the old one.

After sweeping out the entire frame, the guns are reset to the top of the screen again, and this takes a short period of time. This time period is called the vertical retrace, and during this period (much shorter than 60th of a second), nothing can change on the screen. The **swapbuffers** subroutine exchanges the front and back buffers in double buffer mode during the next vertical retrace. The system waits for the vertical retrace so that the currently displayed buffer is completely drawn.

If it did not wait for the vertical retrace, a frame would be drawn partly from one buffer, and partly from another, causing a serious visual disturbance. Because vertical retraces occur every 60th of a second on the standard monitor, the **swapbuffers** subroutine can block the running process for up to that long. (The default monitor is refreshed 60 times per second. Other options can have other retrace periods.)

Once an image is fully drawn in the back buffer, the **swapbuffers** subroutine displays it. This subroutine is ignored in single buffer mode.

> Note: A caution is in order for double buffered programs. Suppose you are writing a flight simulator that draws each frame, and then swaps the buffers. Suppose it runs at a certain rate (say 60 frames per second). As you modify the program to increase the complexity of the scene, eventually you reach a point where the drawing cannot be completed in a 60th of a second.

At this point, because the **swapbuffers** subroutine must wait for a vertical retrace, the frame rate suddenly drops to 30 per second; that is, adding the last polygon cuts the performance in half. There is no smooth degradation. Similarly, as more geometry is added, the rate drops to 20 per second, 15 per second, and so on. Properly tuning such a program can be tricky if smooth motion is required. The **swapinterval** subroutine helps adjust the timing between buffer swaps. Read the discussion on the **swapinterval** subroutine for further information.

The syntax follows:

```
void swapbuffers()
```

gconfig Subroutine

The **gconfig** subroutine sets the modes that you have requested. You must call the **gconfig** subroutine for the **doublebuffer**, **multimap**, **overlay**, **underlay**, **onemap**, **RGBmode**, **cmode**, and **singlebuffer** subroutines to take effect.

After a call to the **gconfig** subroutine, the writemask and color attributes are no longer defined. The contents of the color map do not change. The syntax follows:

```
void gconfig()
```

singlebuffer Subroutine

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. The **singlebuffer** subroutine does not take effect until the **gconfig** subroutine is called. Single buffer mode is the default. The syntax follows:

```
void singlebuffer()
```

frontbuffer Subroutine

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.

A **frontbuffer** subroutine setting of True enables simultaneous updating of (or writing into) the front while the rear buffer is being updated. Its parameter is a 32-bit integer value. The **gconfig** subroutine sets the value of the **frontbuffer** subroutine to False. This subroutine is useful only in double buffer mode. The syntax follows:

```
void frontbuffer(Int32 bool)
```

backbuffer Subroutine

It is sometimes convenient to update both the front and the back buffers, or to update the front buffer instead of the back. The **backbuffer** subroutine enables updating in the back buffer. Its parameter is a 32-bit integer value. When the value of the parameter is True, the default, the back buffer is enabled for writing. When the value of the parameter is False, the back buffer is not enabled for writing.

The **gconfig** subroutine sets the value of the **backbuffer** subroutine to True. The syntax follows:

```
void backbuffer(Int32 bool)
```

getbuffer Subroutine

The **getbuffer** subroutine indicates which buffer or buffers are enabled for writing in double buffer mode. The values returned can be compared to the values of MSINGLE, MPROJECTION, and MVIEWING to determine the current mode. The back buffer is enabled by default. Other returned values indicate that the front buffer or both buffers are enabled. The **getbuffer** subroutine returns zero if neither buffer is enabled or if the system is not in double buffer mode. The syntax follows:

```
Int32 getbuffer()
```

swapinterval Subroutine

The **swapinterval** subroutine 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 the **swapbuffers** subroutine.

The **swapinterval** subroutine is typically used when you want to show frames at a constant rate, but the images vary in complexity. To achieve a constant rate, the swap interval is set to be long enough that even the most complex frame can be drawn in that time. If a simple frame is drawn, the user's process simply blocks and waits until the swap interval is used up. The default interval is 1.

The **swapinterval** subroutine is valid only in double buffer mode. The syntax follows:

```
void swapinterval(Int16 interval)
```

getdisplaymode Subroutine

The **getdisplaymode** subroutine returns the current display mode. The values returned can be compared to the values of MSINGLE, MPROJECTION, and MVIEWING to determine the current mode. The syntax follows:

```
Int32 getdisplaymode()
```

Values returned are:

DMRGB	Indicates RGB single buffer mode.
DMRGBDOUBLE	Indicates RGB double buffer mode.
DMSINGLE	Indicates color map single buffer mode.
DMDOUBLE	Indicates color map double buffer mode.

gsync Subroutine

The **gsync** subroutine waits for the next vertical retrace. Because this subroutine does not return until vertical retrace begins, the calling process is effectively blocked until that time.

This subroutine is useful for pacing the drawing when in single buffer mode. If the amount of drawing to be done is small, this subroutine can be used to achieve a limited amount of smooth animation in single buffer mode. For high-quality, smooth animation, double buffer mode together with the **swapbuffers** subroutine should be used. The syntax follows:

```
void gsync()
```

Underlay and Overlay Modes

Underlay and overlay modes in GL overlays and underlays are independent frame buffers that lie over and lie under, respectively, the principal frame buffer. They can be drawn into and cleared independently from the main frame buffer. In this way, they provide a convenience to the graphics programmer: color values in the underlays and overlays can be changed without destroying the contents of the main frame buffer. If the picture in the main frame buffer took a long time to draw, and the program needs to put up a quick status indicator, or perform some other temporary communications with the user, the overlay planes may be the ideal place to do so. For example, GL pop-up menus use the overlay planes. When the pop-up menus are erased, the contents of the principal frame buffer are unharmed. In some ways, overlays and underlays are not as powerful as the main frame buffer: they contain fewer bitplanes (2 or 4, depending on the graphics adapter) and thus a more limited selection of colors; they can be used only in color index mode, and they do not support Gouraud shading.

The concept of partitions is in many ways similar to that of overlays. Partitions can be independently cleared and drawn into; their stacking order can be changed, and they can be made invisible. Partitions, unlike overlays, are not separate bitplanes, but are a partitioning of the principal frame buffer; thus their name. For more information on partitions, please refer to Partitions.

You can control information overlaid on top of, and placed underneath, the main color frame buffer by setting the number of bitplanes used for overlay and underlay.

Overlay bitplanes supply additional bits of information at each pixel. You can configure the system to have 0, 2, or 4 overlay bitplanes, depending on the installed adapter. Whenever all the overlay bitplanes contain 0 at a pixel, the color of the pixel from the main color bitplanes is presented on the screen.

If any of the overlay planes contains a nonzero entry (there are three ways for this to happen with two overlay bitplanes: 01, 10, and 11), the overlay value is looked up in a separate color table, and that color is presented instead. The overlay color lookup table behaves exactly like the standard color map, except that the lookup table only has three usable entries.

Underlay bitplanes are similar in concept, in that there are extra bits for each pixel. The values of these extra bits 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 presented. Thus, the underlay color shows up only if there is nothing (the pixel value equals 0) in the standard bitplanes. With two underlay bitplanes, there are four possible underlay colors.

Overlay and underlay planes can be used in single or double buffer mode, and in color map or RGB mode. Overlay bitplanes are useful for such things as menus, construction lines, rubber-banding lines, and so forth. Underlay planes might be used for background grids that appear wherever nothing else is drawn. (See the figure entitled ″Frame Buffer Configuration″ on page 132.)

Many of the same operations are available for operating on overlay or underlay bitplanes as are available for the standard bitplanes in color map mode. The color map subroutines (the **color**, **getcolor**, **getmcolor**, **getmcolors**, **mapcolor**, and **mapcolors** subroutines) affect the overlay and underlay bitplanes if the system is in overlay or underlay mode.

For example, in overlay mode, the **color** subroutine sets the overlay color, the **getcolor** subroutine gets the current overlay color, the **mapcolor** subroutine affects entries in the overlay map, and the **getmcolor**

subroutine reads those entries. In overlay mode, all drawing routines draw into the overlay bitplanes rather than the standard bitplanes. The routines are similarly redefined for underlay mode. Use the **drawmode** subroutine to set the overlay or underlay mode.

To set the number of user-defined bitplanes you want to use for underlay color or overlay color, call the **underlay** and **overlay** subroutines. You cannot use the user-defined bitplanes for overlay and underlay color simultaneously. Call the **gconfig** subroutine after the **overlay** or **underlay** subroutines to activate their settings.

List of Underlay and Overlay Mode Subroutines

color	Sets the current color in color map mode.
drawmode	Chooses a set of bitplanes for drawing.
getcolor	Returns the current color in color map mode.
getmcolor	Gets a copy of the RGB values for a color map entry.
mapcolor	Changes a color map entry to a specified RGB value.
overlay	Sets the number of bitplanes used for overlay.
underlay	Sets the number of bitplanes used for underlay.

Default Configuration

By default, the auxiliary (overlay and underlay) planes are always enabled, for all windows, including the root window. Overlays can be disabled for a window only by the application that created the window (Overlays are disabled by making the subroutine calling sequence `overlay(0); gconfig();`. When overlays are disabled, their contents are not visible.)

Overlays are by default enabled for the root window. The default has the following results:

1. When an application renders into the overlays while in fullscreen mode, the rendered drawing is visible even if it overlies other windows.
2. Garbage generated by other applications can be left behind in the overlay planes. Since the overlay planes are not automatically cleared when a program terminates, drawings intentionally or unintentionally left behind in the overlay remain visible. You can use the following code fragment to clear garbage left in the overlay planes:

```
{
 fullscrn();      /* Gain access to the entire screen    */
 drawmode(OVERDRAW);  /* Access the overlay planes specifically */
 color(0);       /* The transparent overlay color     */
 clear();
 drawmode(NORMALDRAW); /* Return to normal operation       */
 endfullscrn();     /* Reset clipping to window boundaries  */
 }
```

The **/usr/lpp/GL/examples/clover.c** file contains an example of the prior code segment.

Configuring Underlay and Overlay Planes

Overlays are configured by making the **overlay**(*n*); **gconfig**; subroutine calling sequence , after having set n equal to the number of desired overlay planes. The number of acceptable values for n are extremely limited, and depend on the installed adapter. Refer to the section “Understanding the Adapter” for information about the number of supported overlay planes for a given adapter. This number can also be queried at run time with the **getgdesc** subroutine. When overlays are disabled (equivalently, when zero overlay planes are configured, by calling `overlay(0); gconfig);`), they become invisible. Although the actual bitplanes are still physically present, their actual contents are no longer visible on the screen. For most of the currently supported graphics adapters, the overlay planes are *not* cleared when they are disabled;any data stored in them remains more or less intact.

drawmode Subroutine

The **drawmode** subroutine is used to put the system into overlay or underlay mode, or back into normal mode after drawing into the overlay bitplanes is finished. The OVERDRAW and UNDERDRAW settings in the **drawmode** subroutine put the system into overlay and underlay mode. The NORMALDRAW setting returns the system to the default, where the color subroutines refer to the standard bitplanes. The other settings for this subroutine are PUPDRAW, which sets operations for the pop-up menu, and CURSORDRAW, which sets operations for the cursor. The syntax is as follows:

```
void drawmode(Int32 mode)
```

Alpha Blending Modes

Alpha-blending provides a mechanism for drawing semi-transparent surfaces. With alpha-blending enabled, pixel colors in the frame buffer can be blended in varying proportion with the color of the graphics primitive being drawn. The proportion is referred to as the “transparency” or alpha value. In normal usage, the incoming color is multiplied by alpha, while the existing color is multiplied by one minus alpha, and the two are summed, to determine the resulting color. That is, the blending is linear:

FinalColor = alpha * IncomingColor + (1.0-alpha) * ExistingColor

In this normal mode of operation, the whole can never be more than the sum of the parts, because (alpha + (1.0-alpha)) equals one. Generically, the blending can be more general:

FinalColor = alpha * IncomingColor + beta * ExistingColor

This usage is more rare, and is used only for special purposes. In GL, the blending is done on a per pixel basis; that is, the blending is done for each pixel individually. Blending is supported in RGB mode only; blending in color index mode is not well supported. The blending is performed for the red, green and blue channels individually:

FinalRed = alpha * IncomingRed + beta * ExistingRed

FinalGreen = alpha * IncomingGreen + beta * ExistingGreen

FinalBlue = alpha * IncomingBlue + beta * ExistingBlue

The alpha and beta blending factors can be set with the **blendfunction** subroutine. For more information on this subroutine, please refer to the Graphics Technical Reference.

By its very nature, alpha-blending is order dependent. That is, drawing one semi-transparent polygon after another gives a different visual result than drawing the one before the other. In most cases, it is desirable to draw the image from back to front, drawing the closest semi-transparent polygon last. The graphics system does not automatically order or re-order graphics primitives to be in the right order when alpha-blending is enabled; it is up to the graphics program to order drawing. Note that the z-buffer, while useful for hidden surface removal, does not actually provide any ordering. That is, if a more distant polygon is drawn after a closer, semi-transparent polygon, that more distant polygon does not automatically appear behind the semi-transparent polygon.

An example program illustrating alpha-blending can be found in the **/usr/lpp/GL/examples** file.

For information on specifying the function to be used for alpha blending, see the **blendfunction** subroutine.

For information on setting the current color in RGB mode, see the **c** subroutine.

For information on specifying RGBA color with a single packed 32-bit integer, see the **cpack** subroutine.

For information on defining a new material, light, or lighting model, see the **lmdef** subroutine.

Writemasks and Logical Operations

This section contains information on the following frame buffer update topics:

- Writemasks
- Partitions
- Writemask for the Z-Buffer
- Logical Operation

Various writemasks control what data is stored in each bitplane of the frame buffer by shielding portions of the frame buffer from being written into. In color map mode, the **writemask** subroutine protects specified bitplanes from ordinary drawing routines. In RGB mode, the **RGBwritemask** subroutine performs this function. The **zwritemask** subroutine controls writing into the z-buffer. The **logicop** subroutine specifies the logical operation to use when writing pixels. The **getwritemask** subroutine and **gRGBmask** subroutine allow you to determine the current writemasks.

List of GL Writemask and Logical Operation Subroutines

blendfunction	Specifies the alpha blending ratio.
getwritemask	Returns the current writemask.
gRGBmask	Returns the current RGB writemask.
logicop	Specifies a logical operation for pixel writes.
RGBwritemask	Grants write permission to a subset of available bitplanes (in RGB mode).
wmpack	Specifies an RGBA writemask with a single packed integer.
writemask	Grants write permission to a subset of available bitplanes in color map mode.
zfunction	Specifies the function used for depth comparison.
zsource	Selects depth or color as the source for z comparisons.
zwritemask	Specifies which bits of the z-buffer are written during normal z-buffer operation.

Writemasks

In all cases when the system uses color maps (the standard bitplanes in color map mode, and the overlay and underlay bitplanes), a writemask is available that can limit the drawing into the bitplanes. By default, the writemask is set up so that there are no drawing restrictions, but it is sometimes useful to limit the effects of the drawing routines. Two 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 each other.

The writemask is described in terms of the standard drawing bitplanes, but exactly the same comments are true if the system is in overlay or underlay mode. For this discussion, it is assumed that only 8 of the 12 bitplanes are used, although the discussion applies equally well to different numbers, including 24 bitplanes.

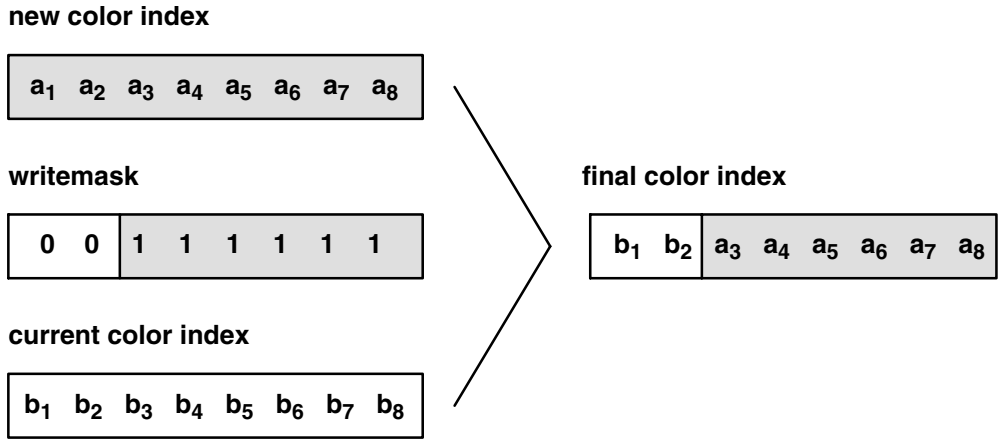
With 8 bitplanes, the color is a number from 0 to 255, which can be represented by 8 binary bits. For example, color 68 is `01000100`. Without writemask controls, if the color is set to 68, then every drawing subroutine puts `01000100` into the 8 bitplanes of the affected pixels.

A writemask restricts this overwriting. If, in the previous example, the writemask were `15 (= 00001111)`, then only the bottom 4 bits of the color are written into the bitplanes (that is, with writemask, a 1 enables a

bitplane for writing, and a 0 disables it). If the color is 68, then any pixels hit by a drawing subroutine would contain `wxyz0100`, where `wxyz` are the 4 bits that were previously there. The 0s (zeros) in the writemask prevent those bits from being overwritten. The default writemask is entirely 1s, so there is no restriction.

Writemasks determine whether a new value can be stored in each bitplane. A 1 (one) in the writemask allows the system to store a new value (0 or 1) in the corresponding bitplane. A 0 (zero) prevents the system from storing a new value, and the corresponding bitplane retains its current value.

In the ″figure″ on page 146, the values in the first and second bits (b1 and b2) do not change because the corresponding positions in the writemask are zero. All the other values (originally b3, b4, ...b8) change to a3, a4, ...a8 because the corresponding positions in the writemask are 1. Each value a1, ...a8 and b1, ...b8 is either 0 or 1, and the setting of the writemask determines whether the value is written.



Writemask Function

Writemask Example Programs

As a very simple example, suppose you wish to draw two completely independent electronic circuits on the screen, power and ground. You would like the power grid to be drawn in blue, the ground grid to be drawn in black, and short-circuits (where both power and ground appear) to be drawn in red. The background color is white.

Initialize the program as follows:

```
#define BACKGROUND  0       /*=00*/
#define POWER       1       /*=01*/
#define GROUND      2       /*=10*/
#define SHORT       3       /*=11*/
mapcolor(0, 255, 255, 255); /*white*/
mapcolor(1, 0, 0, 255);     /*blue*/
mapcolor(2, 0, 0, 0);       /*black*/
mapcolor(3, 255, 0, 0);     /*red*/
```

Then draw all the power circuitry into bitplane 1 and the ground circuitry in bitplane 2. Where both power and ground appear, there is a 1 in both bitplanes, making color 3.

To clear the window before drawing:

```
writemask(3);
color(BACKGROUND);
clear();
```

To draw power circuitry without affecting ground circuitry:

```
writemask(1);
color(1);
<drawing subroutines>
```

To draw ground circuitry without affecting power circuitry:

```
writemask(2);
color(2);
<drawing subroutines>
```

To erase all power circuitry:

```
writemask(1);
color(0);
clear();
```

To erase all ground circuitry:

```
writemask(2);
color(0);
clear();
```

The **circuit.c** example program demonstrates the use of writemasks by drawing power circuitry as previously discussed.

writemask Subroutine

The **writemask** subroutine grants write permission to available bitplanes. It protects bitplanes, in the current drawing mode, that are reserved for special uses from ordinary drawing subroutines. The parameter is a mask with 1 bit available per bitplane.

Whenever there are 1s in the writemask, the corresponding bits in the color index are written into the bitplanes. Zeros in the writemask mark bitplanes as read-only. These bitplanes are not changed, regardless of the bits in the color index.

If the drawing mode is NORMALDRAW, the writemask affects the standard bitplanes; if the mode is OVERDRAW, the writemask affects the overlay bitplanes; if the mode is UNDERDRAW, the writemask affects the underlay bitplanes. Use the **RGBwritemask** subroutine in RGB mode.

It is very important to understand that with the writemask, the bit pattern at each pixel is additive. This means that although you can protect certain bits from being overwritten, all the bits at any pixel are still taken as a single integer or color index value. The syntax is as follows:

```
void writemask(Colorindex writem)
```

getwritemask Subroutine

The **getwritemask** subroutine returns the current writemask of the current drawing mode. The writemask is an integer with up to 12 significant bits, one for each available bitplane. Use the **gRGBmask** subroutine in RGB mode. The syntax is as follows:

```
Int32 getwritemask()
```

RGBwritemask Subroutine

The **RGBwritemask** subroutine is the same as the **writemask** subroutine, except it functions in RGB mode. The *red*, *green*, and *blue* parameters 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. The syntax is as follows:

```
void RGBwritemask(Int16 red, Int16 green, Int16 blue)
```

gRGBmask Subroutine

The **gRGBmask** subroutine returns the current RGB writemask as three 8-bit masks. This subroutine places masks in the low-order 8 bits of the locations pointed to by the *redmask*, *greenmask*, and *bluemask* parameters. The system must be in RGB mode when the **gRGBmask** subroutine executes. The syntax is as follows:

```
void gRGBmask(Int16 *redmask, Int16 *greenmask, Int16 *bluemask)
```

Partitions

Partitioning is a method of slicing the frame buffer into a number of smaller subbuffers. Each subbuffer, or *partition*, has properties and a behavior like the main frame buffer, although not exactly. In some ways, partitions behave like overlays and underlays, and present characteristics of each as follows:

- Any partition can be cleared and drawn into independently, without disturbing the contents of other partitions.
- Partitions can be turned off and on (that is, made hidden or visible) without having to clear or redraw their contents.
- Colors used in any partition can be changed at any time without having to clear or redraw the contents.

Unlike overlays and underlays, partitions can be stacked in any order, and the stacking order can be changed dynamically, without having to redraw any geometry.

A limitation of partitions is that there is a limited number of colors, with the following corollaries:

- No, or limited, smooth (Gouraud) shading.
- No RGB-style (direct color) drawing. Only color index operation is supported.

The foundation of the partition is the writemask. Writemasks are used to protect one set of bitplanes while writing into another set. The concept of partitions can be actualized by careful choice of colors, writemasks, and color maps.

Partitions are created by allocating bitplanes from the main frame buffer. A partition can be one or more bitplanes broad, up to the breadth of the frame buffer. Partitions are protected from one another by using writemasks. This protection mechanism allows the application developer to clear and draw into one partition while leaving the contents of other partitions alone. The stacking order of partitions, and their visibility or invisibility, are determined by the loaded color map.

There are two generic types of partitions, depending on the hardware organization of the color maps and how they are connected to the frame buffer. These types of partitions are referred to here as indexed and component partitions.

There are two types of color map organization: true-color maps and gamma-ramp-type color maps. In a true-color organization, a color index value is stored in the frame buffer. In the gamma-ramp organization, used by the POWERgraphics GTO and Model 730 Supergraphics Processor Subsystem and on the POWER Gt4 and POWER Gt4x, the frame buffer always stores RGB values, but the value of each individual component is passed through a look-up table, traditionally called a gamma ramp. These gamma ramps operate on a per-window basis, not a per-screen basis.

Indexed Partitions

Partitions for true-color-map frame buffer organization use a combination of writemasks and special color maps, set by the user, to reconfigure the frame buffer dynamically into a set of overlay/underlay planes. The total number of bitplanes summed over the partitions must be equal to or less than log2 of the number of color map entries.

Component Partitions

Partitions for gamma-ramp frame buffer organization use the same tools: color maps and writemasks. Instead of groups of bitplanes, however, this method uses sets of RGB triplets.

A basic partition might be two bitplanes reserved out of the red buffer, two from the green buffer, and two from the blue buffer. To determine how many colors can be obtained with this partition, the user can pick one of the three available shades for red and, independently, pick one of the three shades of green and one of the three shades of blue. In this instance, all possible combinations yield 3 x 3 x 3 = 27 different colors on the screen. These colors are not entirely independent of each other; they are mixtures of the component colors, of which there are only three. Therefore, only three totally, truly independent shades are possible.

By judicious choice of the component colors (for example by a least-squares search of the LP space), a large number of visually distinct shades are possible. Thus, the RGB gamma-ramp partition is both limiting and expansive; a 256-entry gamma ramp can be made to display thousands of colors. What looks like two bitplanes can be made to show 27 colors, because there are really 2 x 3 = 6 bitplanes. Three bitplanes can show 7 x 7 x 7 = 343 colors, because there are really 3 x 3 = 9 bitplanes, and so on.

Example source code demonstrating the use of partitions can be found in the **/usr/lpp/GL/examples** directory.

Writemask for the Z-Buffer

The following subroutines control accessibility and comparison values for the z-buffer.

zwritemask Subroutine

The **zwritemask** subroutine controls writing into the z-buffer. The valid settings are `0` (no write at all) and `0xFFFFFF` (write all the bits). This subroutine might be useful for a very complicated background into which a few objects are going to be drawn and moved quickly. Setting the **zwritemask** subroutine to zero locks in the background information and prevents its modification. Whether the new objects are drawn depends on the results of the depth comparison. The syntax is as follows:

```
void zwritemask(Int32 mask)
```

zfunction Subroutine

The **zfunction** subroutine compares the z value of the current context (destination value) of a pixel against the z value for the input (source value) pixel. If the result of the comparison matches the subroutine's parameter, the system draws new values into that pixel. The syntax is as follows:

```
void zfunction(Int32 func)
```

zsource Subroutine

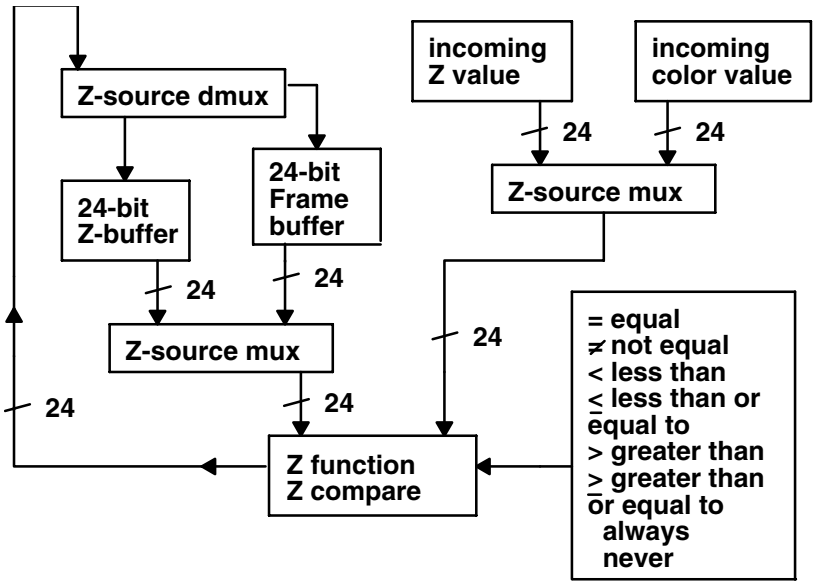
The **zsource** subroutine selects either depth or color as the source for *z* comparisons. After a call to the **gbegin**, **ginit**, **greset**, or **winopen** subroutine, the default z-buffering is done with depth (z) values.

Note: The **zsource** subroutine is unavailable on the POWER Gt4 and POWER GXT1000 adapters.

You can set the source for comparison on color buffers rather than on the z-buffer. This is useful primarily for drawing antialiased lines that cross each other. The syntax is as follows:

```
void zsource(Int32 source)
```

The ″figure″ on page 150 shows the implementation of the z-source function.



Z-source Diagram

Logical Operation

A logical operation determines how the system combines the color for each pixel produced by a primitive with the current color of the destination pixel in the frame buffer.

logicop Subroutine

The **logicop** subroutine specifies the bit-wise logical operation for writing pixels. The logical operation is applied between the incoming (source) and existing (destination) values to generate the final pixel value. In color map mode, all writemask-enabled index bits (up to 12) are changed. In RGB mode, all enabled component bits (up to 24) are changed.

The **logicop** subroutine is valid in all drawing modes (NORMALDRAW, UNDERDRAW, OVERDRAW, PUPDRAW, and CURSORDRAW) and in both color map and RGB modes. This subroutine affects all drawing operations, including points, lines, polygons, and pixel area transfers. The setting of the **logicop** subroutine does NOT apply to pixel block transfers to the z-buffer. The syntax is as follows:

```
void logicop(Int32 opcode)
```

You can also read Configuring the Frame Buffer and how frame buffer update relates to Creating Animated Scenes and Removing Hidden Surfaces.

Clearing, Resetting, and Initializing GL

GL contains a number of subroutines that perform startup functions or reconfigure the system to accommodate a requested mode of operation. For example, the **overlay**, **underlay**, **doublebuffer**, **singlebuffer**, **multimap**, **onemap**, **RGBmode**, and **cmode** subroutines do not take effect until the **gconfig** subroutine is called.

Other subroutines initialize or terminate programs (**ginit** and **gexit**); clear the viewport, color bitplanes, and z-buffer (**clear**, **zclear**, and **czclear**); or turn on or off full-screen mode (**fullscrn** and **endfullscrn**). The **greset** subroutine resets all global state attributes to initial values. The **gbegin** subroutine initializes the graphics system without changing the color map. The **gversion** subroutine returns the version of GL being used. The **getgdesc** subroutine returns information about the currently installed graphics hardware.

GL runs on top of Enhanced X-Windows and initializes with a default font. GL also has access to X fonts through the X server.

List of GL Clearing, Resetting, and Initializing Subroutines

clear	Clears to the screenmask.
czclear	Clears the color bitplanes and the z-buffer simultaneously.
endfullscrn	Ends full screen mode.
fullscrn	Enables drawing outside current window boundaries.
gbegin	Initializes the graphics system without changing the color map.
gconfig	Reconfigures the system.
gexit	Terminates a graphics program.
ginit	Initializes the graphics system.
greset	Resets all global state attributes to initial values.
gversion	Returns the version of GL being used.
zclear	Initializes the z-buffer.

Chapter 10. Working with Objects (Display Lists)

This section discusses the following topics on defining, using, and modifying objects in GL:

- List of GL Object (Display List) Subroutines
- Defining an Object
- Using Objects
- Mapping Screen Coordinates to World Coordinates
- Object Editing

It is sometimes convenient to group a sequence of drawing routines and give it an identifier. The entire sequence can then be repeated with a single reference to the identifier rather than by repeating all the drawing routines. In GL, such sequences are called graphical objects; on other systems they are sometimes known as display lists.

A graphical object is a list of graphics primitives (drawing routines) to display. For example, a drawing of an automobile can be viewed as a compilation of smaller drawings of each of its parts: windows, doors, wheels, and so forth. Each part might be a graphical object: a series of calls to the **move**, **draw**, and **pdr** subroutines.

To make the automobile a graphical object, first create objects that draw its parts, such as a wheel object, a door object, and a body object. The automobile object is a series of calls to the part objects, which together with appropriate rotation, translation, and scale routines, puts all the parts in their correct places.

List of GL Object (Display List) Subroutines

bbox2	Culls and prunes to the bounding box.
callobj	Draws an instance of an object.
chunksize	Specifies the minimum object size in memory.
closeobj	Closes an object.
compactify	Compacts memory storage of an object.
delobj	Deletes an object.
deltag	Deletes a tag from an object.
editobj	Opens an object for editing.
genobj	Returns a unique integer for use as object identifier.
gentag	Returns a unique integer for use as tag number.
getopenobj	Returns the current open object.
isobj	Establishes the uniqueness of object number.
istag	Establishes the uniqueness of tag number.
makeobj	Creates a new object (display list).
maketag	Inserts a tag into the display list.
mapw	Maps a point on the screen into a line in 3-D world coordinates.
mapw2	Maps a point on the screen into a line in 2-D world coordinates.
newtag	Inserts a tag at an offset from an existing tag.
objdelete	Deletes a routine from an object.
objinsert	Inserts a routine into an object.
objreplace	Replaces the existing display list routine with a new one.

Defining an Object

Create and name objects with the **makeobj** subroutine. When you call the **makeobj** subroutine, the system defines an object. Its *object* parameter is a signed 32-bit integer, which is the object's numeric identifier.

When the **makeobj** subroutine executes, the system enters the object's numeric identifier into a symbol table and allocates memory for its list of drawing routines. This opens a new, empty object to which you can add drawing routines. When you open an object for editing, drawing routines are not executed and drawn on the screen, but are added to the list until the **closeobj** subroutine is called.

Thus, a graphical object is a list of primitive drawing routines to be executed. Drawing the display list consists of executing each routine in the listed order. There is no flow control, such as looping, iteration, or condition tests (except for the **bbox2** subroutine).

> Note: Not all Graphics Library routines can be put in a display list. A general rule is to include drawing routines and not to include routines that return values. If you have a question about a particular subroutine, check that entry in Choosing the Right GL Subroutine.

makeobj Subroutine

The **makeobj** subroutine creates a graphical object. It takes one parameter, a signed 32-bit integer that is associated with the object. If the *object* parameter is the number of an existing object, the contents of that object are deleted.

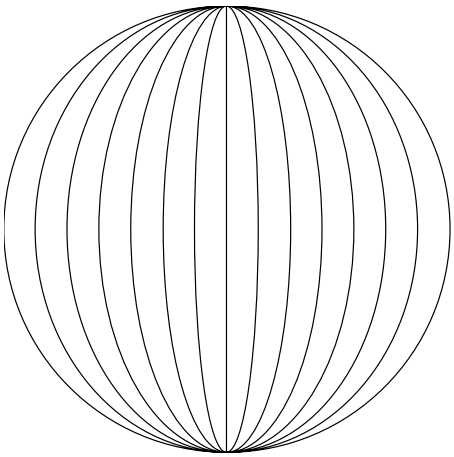
When the **makeobj** subroutine executes, the object number is entered into a symbol table and memory is allocated for a display list. Subsequent graphics routines are compiled into the display list instead of executing. The syntax is as follows:

```
void makeobj(Int32 object)
```

closeobj Subroutine

The **closeobj** subroutine terminates the object definition and closes the open object. All the routines in the graphical object between the **makeobj** and **closeobj** subroutines are part of the object definition.

The following fragment shows the object definition of a simple shape, as illustrated in the following ″figure″ on page 155.



Sphere

```
makeobj(sphere=genobj);
for (phi=0; phi<=PI; phi=PI/9){
     for (theta=0; theta<=2*PI; theta+=PI/18){
          x=sin(theta) * cos(phi);
          y=sin(theta) * sin(phi);
          z=cos(theta);
          if (theta==0)move (x,y,z);
          else draw (x,y,z);
    }
}
closeobj();
```

If you specify a numeric identifier that is already in use, the system replaces the existing object definition with the new one. To ensure your object's numeric identifier is unique, use the **isobj** and **genobj** subroutines.

isobj Subroutine

The **isobj** subroutine tests whether there is an existing object with a given numeric identifier. Its *object* parameter specifies the desired numeric identifier. The **isobj** subroutine returns TRUE if an object exists with the specified numeric identifier, and FALSE if none exists. The syntax is as follows:

```
Int32 isobj(Int32 object)
```

genobj Subroutine

Use the **genobj** subroutine to generate a unique numeric identifier. It does not generate current numeric identifiers. This subroutine is useful in naming objects when it is impossible to anticipate what the current numeric identifier will be when the routine is called. The syntax is as follows:

```
Int32 genobj()
```

delobj Subroutine

The **delobj** subroutine deletes an object. It frees all memory storage associated with the object. The numeric identifier is undefined until it is reused to create a new object. The system ignores calls to deleted or undefined objects. The syntax is as follows:

```
void delobj(Int32 object)
```

Using Objects

After an object has been created, there are several subroutines for using the object in a display.

callobj Subroutine

Once you create an object, use the **callobj** subroutine to draw it on the screen. Its *object* parameter takes the numeric identifier of the object you want to draw. The syntax is as follows:

```
void callobj(Int32 object)
```

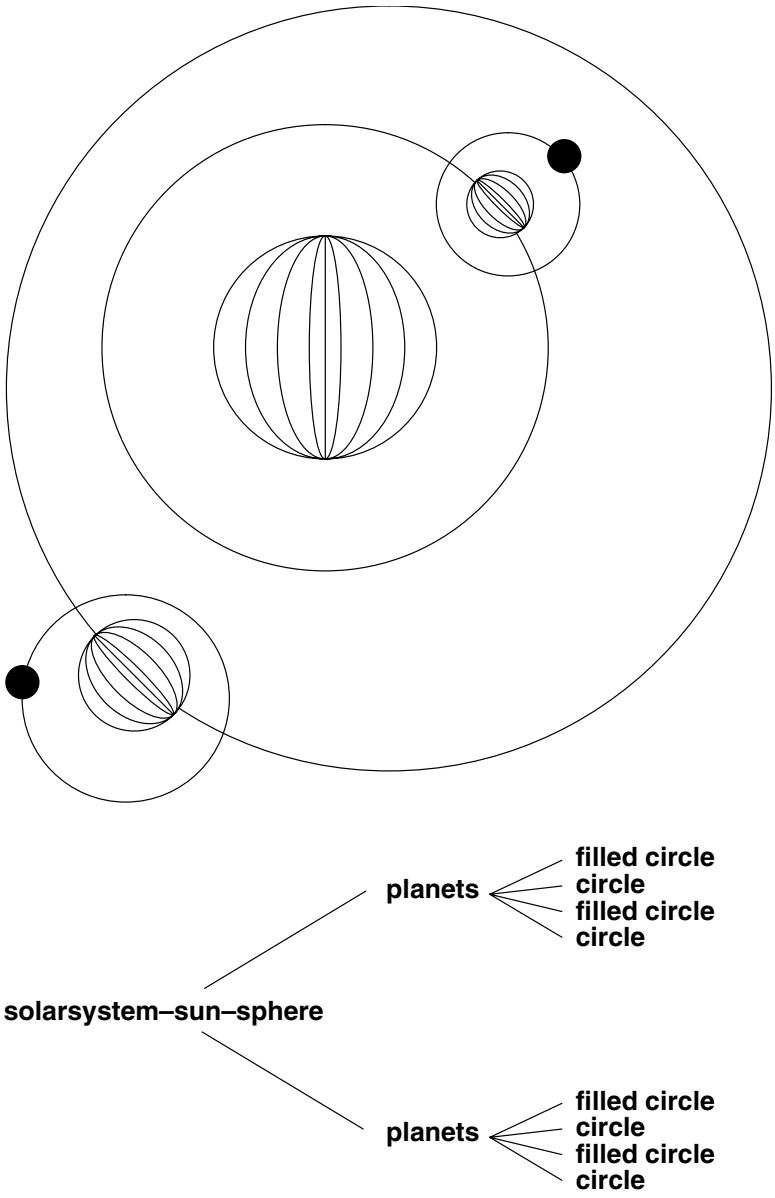
You can use the **callobj** subroutine to call one object from inside another object. You can draw more complex pictures when you use a hierarchy of simple objects. For example, the following program uses a single `callobj (pearl)` call to draw the object, a string of pearls, by calling the previously defined object `pearl` seven times.

```
Int32 pearl = 1, pearls = 2

makeobj(pearl);
   color(BLUE);
   for(angle=0; angle<3600; angle=angle+300) {
      rotate(300, 'y');
      circ(0.0, 0.0, 1.0);
   }
closeobj();

makeobj(pearls);
   for(i=0; i<7; i=i+1) {
      translate(2.0, 0.0, 0.0);
      color(i);
      callobj(pearl);
    }
closeobj();
```

The ″Solarsystem″ on page 157 figure shows another example using simple objects to build more complex ones. It defines a solar system as a hierarchical object. Calling one object `solarsystem` draws all the other objects named in its definition (the sun, the planets, and their orbits).



Solarsystem

Solarsystem, a complex object, is defined hierarchically, as shown in the tree diagram. Branches in the tree represent the **callobj** subroutines.

The system does not save global attributes before the **callobj** subroutine takes effect. Thus, if an attribute, such as color, changes within an object, the change can affect the caller as well. When needed, use the **pushattributes** and **popattributes** subroutines to preserve global attributes.

When a complex object is called, the system draws the whole hierarchy of objects in its definition. For example, in the ″Solarsystem″ on page 157 figure, because the system draws the whole object `solarsystem`, it can draw objects that are not visible in the viewport. The settings in the **bbox2** subroutine determine whether an object is within the viewport and whether it is large enough to be seen.

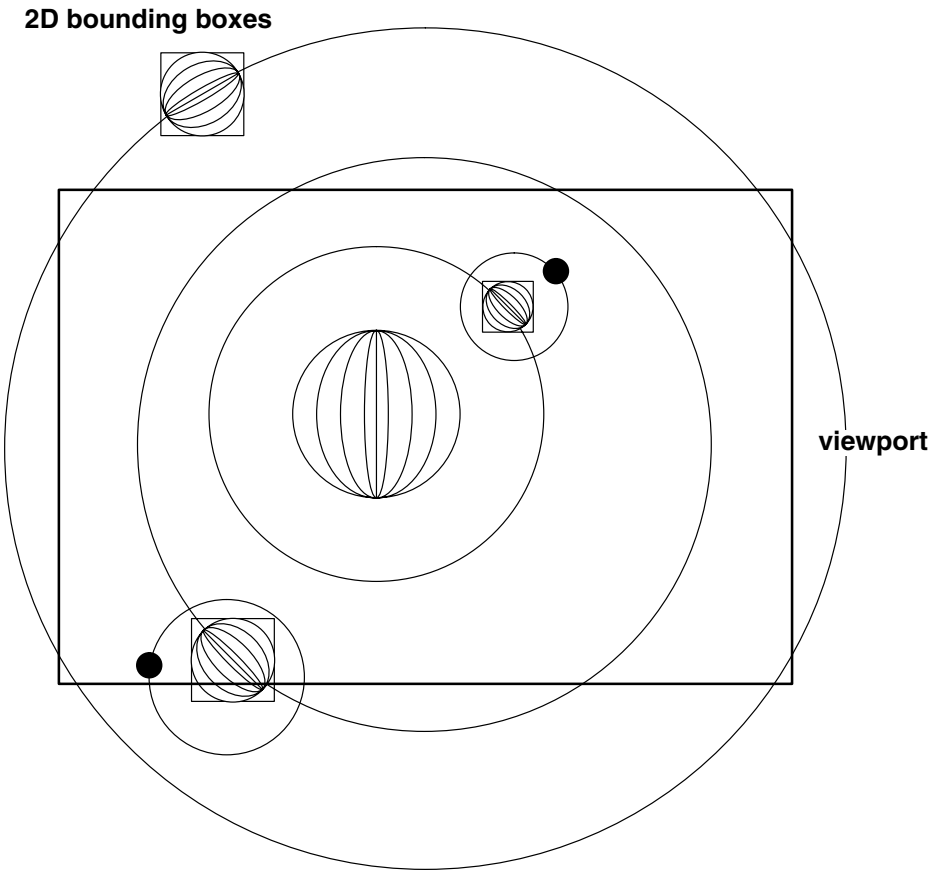
bbox2 Subroutine

The **bbox2** subroutine performs the graphical functions known as pruning and culling. Culling determines which parts of the picture are less than the minimum feature size and, thus, too small to draw on the screen. Pruning calculates whether an object is completely outside the viewport.

The **bbox2** subroutine takes as its parameters an object space bounding box in coordinates (given in the *x1*, *y1*, *x2*, *y2* parameters) and minimum horizontal and vertical feature sizes (given in the *xmin*, *ymin* parameters) in pixels. The system calculates the bounding box, transforms it to screen coordinates, and compares it with the viewport.

If the bounding box is completely outside the viewport, the routines between the **bbox2** subroutine and the end of the object are ignored. If the bounding box is within the viewport, the system compares it with the minimum feature size. If it is too small in both the x and y dimensions, the rest of the routines in the object are ignored. Otherwise, the system continues to interpret the object.

The ″Bounding Boxes″ on page 158 figure shows some of the objects within the complex object ″Solarsystem″ on page 157 juxtaposed to specified bounding boxes. The bounding boxes can perform pruning. to determine what objects are partially in the viewport.



Bounding Boxes

Bounding boxes are computed to determine which objects are outside the screen viewport. If the bounding box is entirely outside the viewport, the rest of the object display list is not traversed. The sphere in the bounding box that lies partially within the viewport is drawn and clipped to the edge of the viewport. The syntax is as follows:

```
void bbox2(Screencoord xmin, Screencoord ymin,
           Coord x1, Coord y1, Coord x2, Coord y2)
```

Mapping Screen Coordinates to World Coordinates

The following subroutines map screen coordinates to world coordinates. If you want to know what the world coordinate values are for the screen coordinates of an object, these subroutines provide a 2-D and a 3-D method to perform the mapping.

mapw Subroutine

The **mapw** subroutine takes a 2-D screen point and maps it onto a line in 3-D world space. Its *viewobj* parameter contains the viewing, projection, and viewport transformations that map the current displayed objects to the screen.

The **mapw** subroutine reverses these transformations and maps the screen coordinates back to world coordinates. It returns two points in the *(modelx1*, *modely1*, *modelz1)* and the *(modelx2*, *modely2*, *modelz2)* parameters, which specify the endpoints of the line. The *screenx* and *screeny* parameters specify the screen point to be mapped. The syntax is as follows:

```
void mapw(Int32 viewobj, Screencoord screenx, Screencoord screeny,
          Coord *modelx1, Coord *modely1, Coord *modelz1,
          Coord *modelx2, Coord *modely2, Coord *modelz2)
```

mapw2 Subroutine

The **mapw2** subroutine is the 2-D version of the **mapw** subroutine. In 2-D, the system maps screen coordinates to world coordinates rather than to a line. The *viewobj* parameter contains the projection and viewing transformations that map the displayed objects to world coordinates; the *screenx* and *screeny* parameters define screen coordinates. The *modelx* and *modely* parameters return the corresponding world coordinates. If the transformations in the *viewobj* parameter are not two-dimensional (orthogonal) projections, the result is undefined. The syntax is as follows:

```
void mapw2(Int32 viewobj, Screencoord screenx,
           Screencoord screeny, Coord *modelx, Coord *modely)
```

Object Editing

This section discusses the following topics

- Identifying Display List Items with Tags
- Inserting, Deleting, and Replacing Within Objects
- Object Editing Examples
- Object Memory Management

You can change an object by editing it. Editing requires you to identify and locate the drawing routines that you want to change. You use two types of routines when you edit an object:

- Editing subroutines, which add, remove, or replace drawing routines.
- Tag subroutines, which identify locations of drawing routines within an object.

editobj Subroutine

To open an object for editing, use the **editobj** subroutine. A pointer acts as a cursor that appends new routines. The pointer is initially set to the end of the object. The system appends graphics routines to the object until either a **closeobj** subroutine or a pointer positioning routine (the **objdelete**, **objinsert**, or **objreplace** subroutines) executes. The syntax is as follows:

```
void editobj(Int32 object)
```

The system interprets the editing routines following the **editobj** subroutine call. Use the **closeobj** subroutine to terminate your editing session. If you specify an undefined object, an error message appears.

getopenobj Subroutine

To determine if an object is open for editing, use the **getopenobj** subroutine. If an object is open, it returns the object's numeric identifier. If no object is open, it returns -1. The syntax is as follows:

```
Int32 getopenobj()
```

Identifying Display List Items with Tags

Tags locate display list items you want to edit. Editing routines require tag names as parameters. The STARTTAG value is a predefined tag that goes before the very first item to mark the beginning of the list. The STARTTAG value does not have any effect on drawing or modifying the object; use it only to return to (find) the beginning of the list.

The ENDTAG value is a predefined tag that is positioned after the last item to mark the end of the list. Like STARTTAG, ENDTAG does not have any effect on drawing or modifying the object; use it to find the end of the graphical object. When you call the **makeobj** subroutine to create a list, STARTTAG and ENDTAG automatically appear. You cannot delete these tags. When an object is opened for editing, a pointer appears at ENDTAG, after the last routine in the object. To perform edits on other items, refer to them by their tags.

maketag Subroutine

Use tags to mark display list items you may want to change. You explicitly tag routines with the **maketag** subroutine. You specify a signed 32-bit numeric identifier and the system places a marker between two list items. You can use the same tag name in different objects. The syntax is as follows:

```
void maketag(Int32 tag)
```

newtag Subroutine

The **newtag** subroutine also adds tags to an object, but uses an existing tag to determine its relative position within the object. The **newtag** subroutine creates a new tag that is offset beyond the other tag by the number of lines given in its parameter offset. The syntax is as follows:

```
void newtag(Int32 newt, Int32 oldtag, Int32 offset)
```

istag Subroutine

The **istag** subroutine tells whether a given tag is in use within the current open object. The subroutine returns TRUE if the tag is in use, and FALSE if it is not. The result is undefined if there is no currently open object. The syntax is as follows:

```
Int32 istag(Int32 tag)
```

gentag Subroutine

The **gentag** subroutine generates a unique integer to use as a tag within the current open object. The syntax is as follows:

```
Int32 gentag()
```

deltag Subroutine

The **deltag** subroutine deletes tags from the object currently open for editing. Remember, you cannot delete the special STARTTAG and ENDTAG tags. The syntax is as follows:

```
void deltag(Int32 tag)
```

Inserting, Deleting, and Replacing within Objects

The following subroutines allow you to edit an object by moving the subroutines inside the object using tags as markers for inserting or deleting.

objinsert Subroutine

Use the **objinsert** subroutine to add routines to an object at the location specified in the *tag* parameter. The **objinsert** subroutine positions an editing pointer on the tag specified in the *tag* parameter. The system inserts graphics routines immediately after the tag. To terminate the insertion, use the **closeobj** subroutine or the **objdelete** or **objreplace** editing subroutines. The syntax is as follows:

```
void objinsert(Int32 tag)
```

objdelete Subroutine

The **objdelete** subroutine removes routines from the current open object. It removes everything between the *tag1* and *tag2* parameters, including routines and other tag names. For example, `objdelete(STARTTAG, ENDTAG)` would delete every routine. The system ignores the **objdelete** subroutine if no object is open for editing. This routine leaves the pointer at the *tag1* parameter location after it executes. The syntax is as follows:

```
void objdelete(Int32 tag1, Int32 tag2)
```

objreplace Subroutine

The **objreplace** subroutine combines the functions of the **objdelete** and **objinsert** subroutines. It provides a quick way to replace one routine with another that occupies the same amount of display list space. Its *tag* parameter is a single tag. Graphics routines that follow the **objreplace** subroutine overwrite existing routines until an occurrence of the **closeobj** or an editing subroutine (**objinsert** or **objdelete**) terminates the replacement.

> Note: The **objreplace** subroutine requires that the new routine to be exactly the same length in bytes as the previous one. If it is not, there is a danger that the display list will be scrambled. Use the **objdelete** and **objinsert** subroutines for more general replacement.

The syntax is as follows:

```
void objreplace(Int32 tag)
```

Object Editing Examples

The following is an example of object editing. The object `star` is defined.

```
makeobj(star);
   color(GREEN);
   maketag(BOX);
   recti(1, 1, 9, 9);
   maketag(INNER);
   color(BLUE);
   poly2i(8, Inner);
   maketag(OUTER);
   color(RED);
   poly2i(8, Outer);
   maketag(CENTER);
   color(YELLOW);
   pnt2i(5, 5);
closeobj();
```

This object is then edited with the following routine to produce a modified object:

```
editobj(star);
   circi(1, 5, 5);
   objinsert(BOX);
```

```
   recti(0, 0, 10, 10);
   objreplace(INNER);
   color(GREEN);
closeobj();
```

The object resulting from the editing session is equivalent to an object created by the following code.

```
makeobj(star);
   color(GREEN);
   maketag(BOX);
   recti(0, 0, 10, 10);
   recti(1, 1, 9, 9);
   maketag(INNER);
   color(GREEN);
   poly2i(8, Inner);
   maketag(OUTER);
   color(RED);
   poly2i(8, Outer);
   maketag(CENTER);
   color(YELLOW);
   pnt2i(5, 5);
   circi(1, 5, 5);
closeobj();
```

Object Memory Management

Editing can require large amounts of memory. The **compactify** and **chunksize** subroutines perform memory management tasks.

compactify Subroutine

As memory is modified by the various editing routines, an open object can become fragmented and stored inefficiently. When the amount of wasted space becomes large, the system automatically calls the **compactify** subroutine during the **closeobj** operation. The routine allows you to perform the compaction explicitly. Unless new routines are inserted in the middle of an object, compaction is not necessary.

> Note: The **compactify** subroutine uses a significant amount of computing time. Do not call it unless the amount of available storage space is critical; use it sparingly when performance is a consideration.

The syntax is as follows:

```
void compactify(Int32 object)
```

chunksize Subroutine

The **chunksize** subroutine specifies the minimum chunk of memory necessary to accommodate the largest GL command call. Normally, this is a call to the **poly** or **polf** subroutine with a very large number of vertices. If there is a memory shortage, use the **chunksize** subroutine to allocate memory differently to an object.

The **chunksize** subroutine specifies the minimum amount of memory that the system allocates to an object. The default chunk is 1020 bytes. When a chunk is specified, its size varies according to the needs of the application. As the object grows, more memory is allocated in units of the size specified in the *chunk* parameter. The **chunksize** subroutine is called once after the **ginit** or **winopen** subroutine, and once before the first **makeobj** subroutine.

The **chunksize** subroutine helps use memory economically. For example, when graphical objects require very little memory, the system can be used more efficiently by specifying smaller chunks of memory. There are drawbacks to the use of the **chunksize** subroutine. There is both memory and execution time overhead associated with each chunk. Many small chunks can be inefficient in both ways. The syntax is as follows:

```
void chunksize(Int32 chunk)
```

Chapter 11. Picking and Selecting

GL provides two related mechanisms for returning information about where primitives are being drawn:

- List of GL Picking and Selecting Subroutines
- Picking
- Selecting

Picking returns all primitives that are currently being drawn in the vicinity of the cursor. Selecting returns all primitives that are being drawn into a 3-D rectilinear volume in world space. Both methods return the same type of data and use the same general mechanism for their operation. They differ in the way that the picking region or selecting volume is specified.

There are three concepts behind picking/selecting:

- The picking/selecting region is the area of the screen (for picking) or of world space (for selecting) that is sensitized.
- The format of the returned data is controlled by the name stack and the routines that manipulate the name stack.
- The method of structuring a program to make use of picking/selecting.

List of GL Picking and Selecting Subroutines

endpick	Turns off picking mode.
endselect	Turns off selecting mode.
gselect	Turns selecting mode on.
initnames	Initializes the name stack.
loadname	Loads a name on top of the name stack.
pick	Puts the system in picking mode.
picksize	Sets the dimensions of the picking region.
popname	Pops a name off the name stack.
pushname	Pushes a new name onto the name stack.

Picking

This section discusses the following aspects of picking:

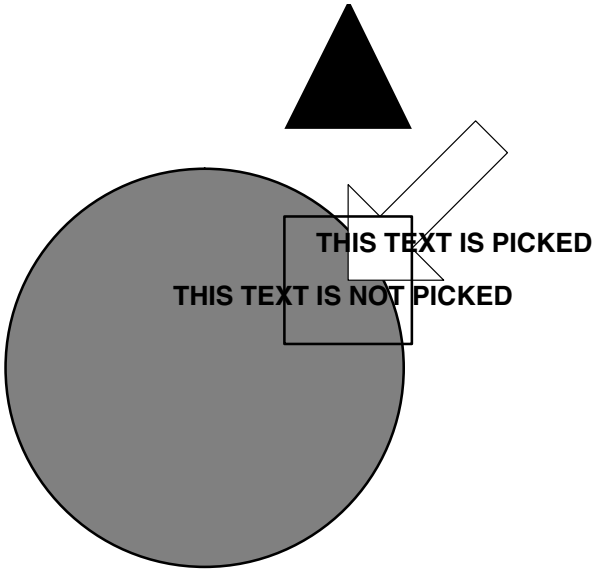
- Picking Introduction
- Recording Hits
- Using the Name Stack
- Defining the Picking Region
- Pick Matrix

Picking Introduction

Use picking to identify the figures (drawing primitives) on the screen that appear near the cursor. To use picking, your software must be structured so that you can regenerate the picture on the screen whenever picking is required. When it is, set the system into picking mode with the **pick** subroutine, redraw the image on the screen, and finish by calling the **endpick** subroutine. Data recorded during the pick appears in the buffer specified by the **pick** and **endpick** subroutines.

When the system is in picking mode, it does not draw anything to the screen. Instead, it checks for *hits*. A hit occurs every time a drawing primitive intersects the picking region. The picking region is a rectangular area of the screen, centered about the location of the cursor. By default, it is 10 by 10 pixels in size. The size of the picking region can be controlled with the **picksize** subroutine.

With one exception, all the standard drawing routines cause hits, including those for points, lines, polygons, arcs, circles, curves, and patches. Raster objects, such as character strings and pixels drawn with the **charstr** subroutine, do not cause hits, but the **cmov** subroutine does. Thus, to pick the string, the cursor must be near the lower left corner of the string. Note also that because the **readpixels** and **readRGB** subroutines are often preceded by a call to the **cmov** subroutine, these routines can appear to cause hits. The following ″figure″ on page 166 illustrates the picking process.



Picking

In picking mode, you can identify the parts of an image that lie near the cursor. The cursor is shown as an arrow. The small box at the tip of the arrow is the picking region. The large shaded circle is picked. The text string whose origin is in the picking region is also picked. The shaded triangle and the other text string are not picked.

Recording Hits

The system records hits by writing data into the picking buffer. The actual data that is recorded is the entire contents of the name stack, preceded by the size of the name stack. The name stack is a stack of 16-bit integers (here referred to as *names*, not to be confused with the actual GL names of the routines that cause hits).

The application (your program) has complete control over the name stack. Names can be loaded onto the stack; pushed onto the stack; popped off the stack with the **loadname**, **pushname**, or **popname** subroutine; or initialized with the **initnames** subroutine.

Note that the actual ASCII name of the drawing routine (for example, **arc**) that caused the hit is *not* recorded. Rather, you must add 16-bit integer names to the name stack or delete them from the name stack to receive interpretable data back when picking is completed.

It is very important to realize that not every hit is recorded. A hit is recorded only if the name stack has been changed since the last hit. The three name stack subroutines, **loadname**, **pushname**, and **popname**, all touch the name stack. Thus, multiple hits can occur, but only one gets recorded into the buffer if the name stack never changes.

For example, suppose your application draws three points widely spaced on the screen, and you want to find which one is close to the cursor using picking mode. Your point-drawing code (that is executed both to draw points and to redraw them in a picking operation), might look like this:

```
ortho(<ortho parameters>);
lookat (<lookat parameters>);
translate (-x, -y, -z);
rotate(30, 'y');
translate(x, y, z);
loadname(0);
pnt(<point 0>);
loadname(1);
pnt(<point 1>);
loadname(2);
pnt(<point 2>);
```

Note that the complete specification for drawing the picture must be there, including any viewing and transformation routines. When this code segment is executed in picking mode and the cursor is near point 1, the buffer returned after the **endpick** subroutine would contain the name 1. If the cursor is near point 2, the buffer would contain the name 2. If the cursor is not near any of the points, an empty buffer would be returned.

The stack is intended for used with hierarchical drawings. For example, suppose you want to draw a car with four instances of a wheel, with each wheel having five instances of a bolt, and you want to pick an individual bolt from the picture. You might have one piece of code to draw each wheel that contained the sequence:

```
pushname(0);
<draw bolt 0>
popname();
pushname(1);
<draw bolt 1>
popname();
   .
   .
   .
```

The car-drawing code might look like this:

```
loadname(0);
<translate>
<draw wheel>
loadname(1);
<translate>
<draw wheel>
   .
   .
   .
```

Each hit on a bolt would occur with the name stack containing two names, the first of which is the wheel number and the second of which is the bolt number on that wheel. Deeper nesting of the hierarchy is possible.

The names reported on hits are completely application dependent. Many drawing routines can occur between changes to the name stack. If any of those routines cause a hit, the contents of the name stack is reported.

Because the contents of the name stack is reported only when it changes, one hit is reported no matter how many of the drawing routines actually draw something near the cursor. If more accuracy than this is required by the application, it must touch the name stack more often. In the code fragment that follows, if all three points caused hits, three identical name stacks are reported:

```
loadname(1);
pnt(-);
loadname(1);
pnt(-);
loadname(1);
pnt(-);
```

pick Subroutine

The **pick** subroutine puts the system in picking mode. The *bufferlen* parameter specifies the length of the buffer array. It should be less than or equal to the size of the buffer as measured in 16-bit short integers. Graphical items that intersect the picking region cause hits. If the name stack has changed since the last hit, the length and contents of the name stack are recorded in the buffer. The syntax is as follows:

```
void pick(Int16 buffer, Int32 bufferlen)
```

Using the Name Stack

You maintain the name stack with the **loadname**, **pushname**, **popname**, and **initnames** subroutines. Each name in the name stack is 16 bits long. You can store up to 1000 names in the name stack. You can intersperse these routines with drawing routines, or you can insert them into display lists.

loadname Subroutine

The **loadname** subroutine puts a new *name* at the top of the name stack and erases what was there before. The syntax is as follows:

```
void loadname(Int16 name)
```

pushname Subroutine

The **pushname** subroutine puts a new *name* at the top of the stack and pushes all the other names in the stack one level lower. The syntax is as follows:

```
void pushname(Int16 name)
```

Before the first **loadname** subroutine is called, the current name is unpredictable. Calling the **pushname** subroutine before calling the **loadname** subroutine can cause unpredictable results.

popname Subroutine

The **popname** subroutine discards the name at the top of the stack and moves all the other names up one level. The syntax is as follows:

```
void popname()
```

initnames Subroutine

The **initnames** discards all the names in the stack and leaves the stack empty. The syntax is as follows:

```
void initnames()
```

endpick Subroutine

The **endpick** subroutine takes the system out of picking mode and returns the number of times the name stack was dumped into the buffer. If this number is positive, the buffer was large enough to contain all of the name stacks written to it. If the number is negative, the buffer was too small to store all the name lists. The magnitude of the returned number is the number of name stacks that were recorded. The syntax is as follows:

```
Int32 endpick(Int16 buffer[])
```

The *buffer* parameter contains copies of the name stack that were recorded as hits occurred. As explained previously, not every hit causes the name stack to be recorded; only the first hit after the name stack has

been touched is recorded. When stored in the buffer, each name stack is preceded by the length of the name stack. If the name stack is empty when a hit occurs, the length is recorded as 0 (zero), and the stack is not recorded.

Defining the Picking Region

Picking loads a projection ″matrix″ on page 169 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 the **pick** subroutine to ensure that the system maps the objects to be picked to the proper coordinates.

$$\text{Picking matrix} = \begin{bmatrix} \frac{vpsizex}{picksizex} & 0 & 0 & 0 \\ 0 & \frac{vpsizey}{picksizey} & 0 & 0 \\ \frac{vpcenterx - cpx}{picksizex} & \frac{vpcentery - cpy}{picksizey} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where:

cpx = cursor position (x)
cpy = cursor position (y)
vpsizex = viewport width
vpsizey = viewport height
vpcenterx = viewport center (x)
vpcentery = viewport center (y)
picksizex = width of the pick rectangle
picksizey = height of pick rectangle

If no projection transformation was originally issued, you must specify the default subroutine, **ortho2**. When the transformation routine is restated, the product of the transformation matrix and the picking matrix is placed at the top of the matrix stack. If you do not restate the projection transformation, picking does not work properly. Instead, the system typically picks every object, regardless of cursor position and picksize.

picksize Subroutine

The default height and width of the picking region is 10 pixels, centered at the cursor. You can change the picking region with the **picksize** subroutine. The *deltax* and *deltay* parameters specify a rectangle centered at the current cursor position (the origin of the cursor glyph). (See Creating a Cursor for a discussion of cursors.) The syntax is as follows:

```
void picksize(Int16 deltax, Int16 deltay)
```

Picking Example Program

The following example program draws an object consisting of three shapes; then it loops, until the right mouse button is pressed. Each time the middle mouse button is pressed, the graphics system:

1. Enters pick mode.
2. Calls the object.
3. Records hits for any routines that draw into the picking region.
4. Prints out the contents of the picking buffer.

Note: When you call an object in picking mode, the screen does not change. Because the picking matrix is recalculated only when pick is called, the system exits and reenters picking mode to obtain new cursor positions.

When the program is run, there are five possible outcomes 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. hit: (2 21)
- The top circle is picked = hit count: 1. hit: (2 22)
- Both the top and bottom circles are picked = hit count: 2 hits: (2 21) (2 22)

```
   /***
       pick.c - example of picking
   ***/
#include <gl/gl.h>
#include <gl/device.h>
#define BUFSIZE 50
void
drawit()
{
   color(RED);
   loadname(1);
   rectfi(20,20,100,100);
   loadname(2);
   pushname(21);
   circi(50, 500, 50);
   popname();
   pushname(22);
   circi(50, 530, 60);
   popname();
}
int
main()
{
short dev, val;
short buffer[BUFSIZE];
int hits;
int xsize, ysize;
int i;
   prefsize(600, 600);
   (void) winopen("pick");
   getsize(&xsize, &ysize);
   color(BLACK);
   clear();
   qdevice(LEFTMOUSE);
   qdevice(ESCKEY);
   for (i = 0;  i < BUFSIZE;  i++)  buffer[i] = 0;

   drawit();
   while (1)  {
   dev = qread(&val);
   switch (dev)  {
      case LEFTMOUSE:
      if (val == 0)  break;
      pick(buffer, BUFSIZE);
      ortho2(-0.5, xsize + 0.5, -0.5, ysize + 0.5);
      drawit(); /* no actual drawing takes place */
      hits = endpick(buffer);
```

```
      /* display hit information */
      {
      int index, items, h, i;
      printf("hit count: %d   hits:  ", hits);
      index = 0;
      for (h = 0;  h < hits;  h++)  {
         items = buffer[index++];
         printf("(");
         for (i = 0;  i < items;  i++)  {
            if (i != 0)  printf(" ");
            printf("%d", buffer[index++]);
            }
         printf(") ");
         }
      printf("\n");
      }
      break;
      case ESCKEY: /* exit program */
      return 0;
      break;
      }
   }
}
```

Pick Matrix

The **pick** and **endpick** subroutines create the following ″matrix″ on page 169 and load it as the current matrix. This formula is placed here for information only. It is not necessary to understand its meaning to use it successfully.

Selecting

Selecting is much like picking, but is more general. The same concept of hits is used, and the same name stack manipulation routines apply. The returned data buffer is filled in the same way. The only difference is in the way the selecting region is defined.

The selecting region corresponds to a rhomboid in world coordinates. Any drawing primitive that intersects with this rhomboid causes a hit and is recorded in the same manner as in picking. There is no special call to set the selecting volume; instead, the matrix manipulation routines are used.

Like picking, no actual drawing occurs while the system is in select mode. Instead, all drawing primitives are transformed through the geometry pipeline and compared to the canonical 3-D unit cube clipping volume. If the drawing primitive is inside this volume, it causes a hit. If it is wholly outside, there is no hit recorded. Transforming through the pipeline means that every drawing primitive is run through the combination of the modeling, viewing, and projection transformations. If, after these transformations, any portion of the primitive ends up inside the volume $-w <= x, y, z <= +w$, a hit has occurred.

To perform selecting, you must set up your viewing/projection matrices correctly. You can set up the matrices either immediately before or after a call to the **gselect** subroutine. The **gselect** subroutine, unlike the **pick** subroutine, does not alter the matrix stack.

In most applications, only the viewing matrix is changed, not the projection matrix. The viewing matrix controls what portion of the world space ends up inside the NDC unit cube ($-w <= x, y, z <= +w$). Also, the individual modeling routines are not changed because they only serve to embed individual subportions of the drawing into the world coordinate frame. The usual process is to draw the same primitives in the same places they previously occupied while selecting is going on.

To end selecting mode, call the **endselect** subroutine. The returned data is in exactly the same format as for the **endpick** subroutine.

gselect Subroutine

The **gselect** subroutine turns on the selection mode. The **gselect** and **pick** subroutines are identical, except that the **gselect** subroutine is more general. It allows you to define an arbitrary rhomboid in world coordinates as the selecting region. The syntax is as follows:

```
void gselect(Int16 buffer[], Int32 numnames)
```

The *numnames* parameter specifies the maximum number of values that the buffer can store. Each drawing routine that intersects the selecting region causes a hit. The contents of the name stack, preceded by the length of the name stack, are written into the buffer only if this is the first hit since the last time that the name stack was touched. The name stack is controlled in the same way as in picking.

endselect Subroutine

The **endselect** subroutine takes the system out of selecting mode mode and returns the number of times the name stack was dumped into the buffer. If this number is positive, the buffer was large enough to contain all of the name stacks written to it. If the number is negative, the buffer was too small to store all the name lists. The magnitude of the returned number is the number of name stacks that were recorded. The syntax is as follows:

```
Int32 endselect(Int16 buffer[])
```

The *buffer* parameter contains copies of the name stack that were recorded as hits occurred. As explained previously, not every hit causes the name stack to be recorded; only the first hit after the name stack has been touched is recorded. When stored in the buffer, each name stack is preceded by the length of the name stack. If the name stack is empty when a hit occurs, the length is recorded as 0 (zero), and the stack is not recorded.

Selecting Example Program

The following program demonstrates a simple application of selecting. This program draws a planet and then draws a box representing the ship each time the left mouse button is pressed. The program prints `CRASH` and exits when the ship collides with the planet:

```
/***
   crash.c - example of selecting
***/

#include <gl/gl.h>
#include <gl/device.h>
#define BUFSIZE 50
#define PLANET 109
#define SHIPWIDTH 20
#define SHIPHEIGHT 10

void
drawplanet()
{
   circfi(200, 200, 20);
}

int
main()
{
short dev, val;
short buffer[BUFSIZE];
int count, i;
float shipx, shipy, shipz;
int xorigin, yorigin;

   minsize(300, 300);
   (void) winopen("crash");
   getorigin(&xorigin, &yorigin);
```

```
   qdevice(LEFTMOUSE);
   qdevice(ESCKEY);
   color(BLACK);
   clear();
   for (i = 0;  i < BUFSIZE;  i++)  buffer[i] = 0;
   color(RED);
   drawplanet();
   while (TRUE)  {
    dev = qread(&val);
    switch (dev)  {
      case LEFTMOUSE:
         if (val)  {
            shipz = 0;
            shipx = getvaluator(MOUSEX) - xorigin;
            shipy = getvaluator(MOUSEY) - yorigin;
            color(BLUE);
            rect(shipx, shipy,
            shipx + SHIPWIDTH, shipy + SHIPHEIGHT);
            /* specify the selecting region to be a box
            surrounding the rocket ship */
            pushmatrix();
            ortho2(shipx, shipx + SHIPWIDTH,
            shipy, shipy + SHIPHEIGHT);
            initnames();
            gselect(buffer, BUFSIZE); /*enter selecting mode*/
            loadname(PLANET);
            drawplanet(); /* no actual drawing takes place */
            count = endselect(buffer); /* exit select mode */
            popmatrix();
            /* check to see if PLANET was selected */
            if ((count > 0) && (buffer[0] == 1) &&
                  (buffer[1] == PLANET)  )  {
               printf("CRASH!\n");
            }
         }
         break;
     case ESCKEY:
        return 0;
        break;
    }
   }
}
```

Chapter 12. Understanding Windows and Input Control

The Graphics Library provides subroutines to control windows and input from an application program. These subroutines cover the following aspects of window and input control:

- Creating and Managing Windows
- Creating a Cursor
- Using the Keyboard
- Controlling Queues and Devices
- Querying the System in GL
- Creating and Managing Pop-up Menus
- Working with the Textport

Creating and Managing Windows

This section discusses the following aspects of window management:

- Opening and Closing Windows
- Setting Window Attributes and Constraints
- Controlling Window Placement
- Changing Windows Noninteractively
- Managing Multiple Windows
- Other Window Subroutines

GL provides subroutines to create and manipulate windows from an application program. The windowing subroutines are implemented on top of Enhanced X-Windows and work with any window manager, including the AIXwindows window manager. A *GL window* is one in which a GL application draws an image, while a *text window* runs a shell.

List of GL Window Subroutines

blankscreen	Turns screen refresh on and off.
blanktime	Sets screen blanking timeout.
endfullscrn	Ends full screen mode.
fudge	Specifies pixel values to be added to a window size.
fullscrn	Enables drawing outside current window boundaries.
getorigin	Returns the position of a window.
getsize	Returns the size of a window.
iconsize	Specifies the size of a window icon.
icontitle	Specifies the title of a window icon.
keepaspect	Specifies the aspect ratio of a window.
maxsize	Specifies the maximum size of a window.
minsize	Specifies the minimum size of a window.
noborder	Removes the border from a window.
noport	Specifies that a program does not require a window.
prefposition	Constrains the window position and size.
prefsize	Constrains the window size.

stepunit	Specifies a window size change in discrete steps.
swinopen	Creates a restricted subwindow.
winclose	Closes a window.
winconstraints	Binds window constraints to the current window.
windepth	Indicates the stacking order of windows on the screen.
winget	Returns the identifier of the current window.
winmove	Moves the current window by its lower-left corner.
winopen	Creates a new window.
winpop	Raises the current window on top of all other windows.
winposition	Changes current location and size of a window.
winpush	Lowers the current window beneath all other windows.
winset	Sets the current window.
wintitle	Adds a title bar to the current window.

Opening and Closing Windows

The GL subroutines for opening and closing windows are the **winopen**, **swinopen**, and **winclose** subroutines. (The **swinopen** subroutine creates subwindows from a parent window.) Exiting a program causes any existing windows to close automatically.

Setting Window Attributes and Constraints

You can control the size, location, and shape of windows from a GL client program. Calling the **winopen** subroutine without specifying any of these characteristics allows you to open a window of any size or shape anywhere on the screen. You can, however, open a specific window, such as a small square with a border, if you specify the desired size and shape.

Use the window constraint subroutines in the following table to specify window characteristics. Call these subroutines before opening a window with the **winopen** subroutine. GL applies these constraints when it opens the window.

Window Constraint Subroutines	
To Specify	**Use**
Minimum size	**minsize**
Maximum size	**maxsize**
Aspect ratio	**keepaspect**
Size, in pixels	**prefsize**
Size and location	**prefposition**
Sizing increment	**stepunit**
Small increase in size	**fudge**
Size of window's icon	**iconsize**
Give a window's icon a title	**icontitle**
Make a title bar for the current window	**wintitle**
No window borders	**noborder**
No screen space needed	**noport**

To respecify constraints for a window that is already open, first call the desired series of window constraint subroutines from the previous table, then call the **winconstraints** subroutine.

When a user interactively changes a window, the window constraints are automatically enforced. Interactively changing a window means using the mouse and keyboard, in cooperation with the currently running window manager, to move, resize, or reshape a window.

The window constraints allow the applications programmer to control and limit how the user resizes and reshapes a window. The **maxsize** subroutine prevents a user from resizing a window to any size larger than the size set by the subroutine parameters. Likewise, the **minsize** subroutine prevents the user from shrinking a window below the specified size.

The **keepaspect** subroutine prevents a user from resizing a window so that the aspect ratio is changed. The **stepunit** subroutine allows the window to be resized only in specified discrete steps. Normally, a window can be resized by arbitrary amounts; in other words, the **stepunit** subroutine usually specifies one pixel.

The **prefsize** and **prefposition** subroutines constrain the size and position of the window. The **prefsize** subroutine prevents a user from resizing the window. The **prefposition** subroutine prevents a user from moving the window. To place a window on the screen in some position, allowing the user afterwards to reposition or resize the window, call the **winconstraints** subroutine after the **winopen** subroutine (or call the **winconstraints** subroutine twice in a row.) The **winconstraints** subroutine clears previous constraints and sets any new ones.

Controlling Window Placement

Ultimately, window placement is controlled by the window manager. The default window manager for your system is the AIXwindows window manager. To control the behavior of this window manager regarding interpretation and placement, edit the **.Xdefaults** file in the user's home directory.

In controlling interpretation of the **prefposition** subroutine parameters, if you add the following line to the **.Xdefaults** file:

```
Mwm*positionIsFrame: True
```

the location of the window is assumed to include the boundary of the window. That is, the screen coordinates are assumed to refer to the lower left-hand corner of the window border. In contrast, if you add the following line to the **.Xdefaults** file:

```
Mwm*positionIsFrame: False
```

the window manager interprets the position as the location of the window's interior, ignoring the window border.

The AIXwindows window manager normally places a window so that it is fully visible on the screen. To open a window partly or wholly off the screen, add the following line to the **.Xdefaults** file:

```
Mwm*positionOnScreen: False
```

Changing the line to read:

```
Mwm*positionOnScreen: True
```

makes the window manager ignore the values specified with the **prefposition** subroutine and place the window so that it is fully visible on the screen.

In controlling placement of a window, if you add the following line to the **.Xdefaults** file:

```
Mwm*interactivePlacement: True
```

a rubber-banded window appears when the **winopen** subroutine is executed. The user can place the window where desired on the screen. If you change the line to read:

```
Mwm*interactivePlacement: False
```

the window is immediately placed on the screen in the specified position when the **winopen** subroutine is executed.

The AIXwindows window manager normally adds an offset to the location of every new window that is opened. This offset is used to keep new windows from piling up at one place. The offset spreads windows evenly over the entire screen. To turn off this behavior, specify

```
Mwm*clientAutoPlace: False
```

When `clientAutoPlace` is turned off, GL windows are mapped on the screen exactly at the location specified with the **prefposition** subroutine.

Changing Windows Noninteractively

The applications programmer can have the same control over windows that an applications user can. A window may be moved, resized, raised, and lowered from the applications program with the subroutines in the following table:

Window Control Subroutines	
To	**Use**
Move and reshape the current window	**winposition**
Move the current window	**winmove**
Lower the current window to bottom	**winpush**
Raise the current window to top	**winpop**
Return the size of a window	**getsize**
Return the origin of the window	**getorigin**
Return depth of window in the window stack	**windepth**

GL performs these operations on the current window, which is set with the **winset** subroutine.

Managing Multiple Windows

Multiple GL windows opened by the same process do not normally share attributes. Every window has its own current color; its own transformation matrix stack; and its own viewport stack, name stack, and attribute stack (which contains attributes such as the current linestyle and linewidth). The action of subroutines such as **swapbuffers**, **RGBmode**, **depthcue**, and **lmbind** is limited to the current window.

The only exceptions to this rule are a few window management subroutines (such as the **winset** subroutine) and the **def** subroutines: **deflinestyle**, **defpattern**, **defrasterfont**, **loadXfont**, **lmdef**, and **makeobj**. If a linestyle, pattern, light, font, or object is defined for one window, it becomes available to all windows opened by the same process. These definitions are referenced by the same index in all such windows. Information is *never* shared between different processes. The previous comments apply only to windows opened by one and the same process.

The exception made for the **def** subroutines is provided as a convenience to the user. If a linestyle has been defined in one window, and the user wants to use it in another, the definition should not have to be respecified in other, possibly numerous, windows. In addition, the **def** subroutine exception allows the system to operate in a more efficient manner. The system does not need to keep track of multiple copies of duplicate information.

Other than these shared definitions, all other attributes that are part of GL have been made deliberately nonsharable across windows. True multiple, independent drawing sessions can therefore be performed in multiple windows. Separate windows do not interfere with one another, and the applications writer does not have to engage in complicated window attribute management. The system maintains the attribute management automatically.

Other Window Subroutines

GL provides other subroutines to control the windows, as shown in the following table:

Miscellaneous Window Subroutines	
To	**Use**
Put the program in screen space	**screenspace**
Put viewport at current window position	**reshapeviewport**
Enable the entire screen for writing	**fullscrn**
End full-screen mode	**endfullscrn**
Return identifier of current window	**winget**
Set the current window	**winset**
Specify length of screen blank timeout	**blanktime**
Control screen refresh, on/off	**blankscreen**

Creating a Cursor

This section discusses the following aspects of cursors:

- Introduction to Cursors
- Defining a New Cursor
- Cross-Hair Cursor
- Cursor Subroutines

List of GL Cursor Subroutines

curorigin	Sets the origin of the cursor.
cursoff, curson	Control cursor visibility.
curstype	Defines the type and size of cursor.
defcursor	Defines the cursor.
getcursor	Returns the cursor characteristics.
setcursor	Sets cursor characteristics.
attachcursor	Couples cursor position to valuator device.

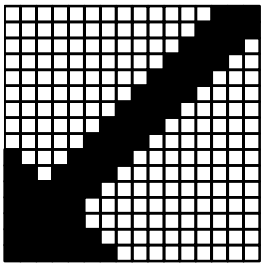
Introduction to Cursors

The cursor is handled with special cursor hardware. As the color guns scan the screen and cross the square region of the screen where the cursor is to be drawn, they look at the corresponding position in the cursor mask to see what color to draw.

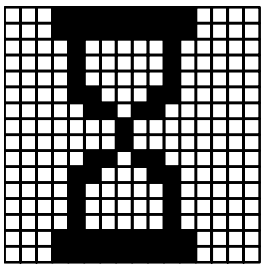
The cursor mask can be 1 or 2 bits deep. If the cursor mask is 0, the normal color is presented. If the mask is nonzero, the mask value is looked up in a color table (very similar to overlay) to find out what

color to draw. The cursor color takes precedence over the overlay color. As with overlays, if the cursor mask is 1 bit deep, there is only one possible color. If the cursor mask is 2 bits deep, the cursor can have up to three colors.

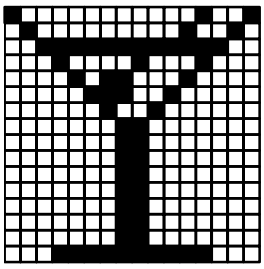
The system supports five different cursor types: a 16-by-16-bit cursor in one or three colors; a 32-by-32-bit cursor in one or three colors; and a cross-hair, one-color cursor. To specify a cursor completely, you need to specify not only its type, but also its shape and colors. In addition, every cursor has an origin, or *hot spot*, and can be turned on or off. See the ″Sample Cursors″ on page 180 figure for examples of a 16-by-16-bit one-color cursor.



```
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  }
```

Sample Cursors

Cursor number zero (0), the default cursor, is an arrow pointing to the upper left corner of the cursor glyph; its origin is at (0, 15), the tip of the arrow. The default cursor cannot be redefined and can always be used. The position of the origin of the cursor is set to the current values of the valuators that are attached to the cursor.

Defining a New Cursor

To define and use a new cursor, you must follow these steps:

1. Enable or disable the visibility of the cursor with the **curson**/**cursoff** subroutines.
2. Set the cursor type to one of the five allowable types with the **curstype** subroutine.
3. Define the cursor's shape and assign it a number with the **defcursor** subroutine.
4. If necessary, define its origin (or hot spot) with the **curorigin** subroutine, and its colors with the **drawmode** and **mapcolor** subroutines.
5. Finally, the new cursor becomes the current cursor with a call to the **setcursor** subroutine.

The following ″figure″ on page 180 illustrates various types of cursors.

If an application needs a number of different cursors, it typically defines all of them on initialization, and then switches from one to another using the **setcursor** (and perhaps. the **mapcolor**) subroutine. Although they do not physically do so, cursors can be thought of as occupying one or two bitplanes of their own, which behave like overlay bitplanes as described previously.

A one-color cursor uses one bitplane and a three-color cursor occupies two. Where there are zeros in the cursor's bitplanes, the contents of the standard, overlay, and underlay bitplanes appear. In the same way that overlay colors are defined, the **drawmode** and **mapcolor** subroutines define the cursor's colors. For example, call the following for a one-color cursor:

```
drawmode(CURSORDRAW);
mapcolor(1, r, g, b);
```

Or, call the following for a three-color cursor:

```
mapcolor(1, r1, g1, b1);
mapcolor(2, r2, g2, b2);
mapcolor(3, r3, g3, b3);
```

When the cursor pattern contains a 1(=01), the color value (r1, g1, b1) is presented. When the cursor pattern is 2(=10), the color value (r2, g2, b2) appears, and so on. Be sure, after you have defined the cursor's colors, to call:

```
drawmode(NORMALDRAW)
```

Cross-Hair Cursor

The cross-hair cursor is formed with two 1-pixel-wide, intersecting lines, one horizontal and one vertical that extend completely across the screen. Its origin is at the intersection of the two lines. This one-color cursor always uses cursor color 3 as its color. The color of the cross-hair cursor is set by mapping color index 1.

The cross-hair cursor consists of a default glyph that cannot be changed. If you assign a value to it with the **defcursor** subroutine, the user-defined glyph is ignored. The cross-hair cursor does not work if more than one window is open.

Cursor Subroutines

The following GL subroutines control cursors.

curson and cursoff Subroutines

The **curson** and **cursoff** subroutines turn the cursor visibility on and off, respectively. However, they execute fairly slowly and should not be heavily used. These subroutines control only the visibility of the cursor and do not disable or enable the cursor or mouse-button click events inside the current window. The **curson** subroutine is the default. The syntax for the **curson** and **cursoff** subroutines is as follows:

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

GL formerly required the cursor to be turned off before any drawing was attempted and turned on again after drawing was completed. This is no longer required. However, existing code that is being ported to the current release may often use this function. This code should be modified to remove these subroutines, or the subroutines should be effectively removed by using `#ifdef` with the C programming language preprocessor. If this is not done, performance will be adversely and severely affected.

curstype Subroutine

The **curstype** subroutine defines the current cursor type. The value given in the *type* parameter is C16X1, C16X2, C32X1, C32X2, or CCROSS. This value is used by the **defcursor** subroutine to determine the dimensions of the arrays that define the cursor's shape. The default value is C16x1. The CCROSS value

indicates a predefined cross-hair cursor, which is one pixel wide. The hot spot is at the center of the cross. Its default center is (15, 15). The CCROSS value uses cursor color 3. The syntax is as follows:

```
void curstype(Int32 type)
```

After you call the **curstype** subroutine, call the **defcursor** subroutine to specify the appropriate-sized array and assign a numeric value to the cursor glyph.

defcursor Subroutine

The **defcursor** subroutine defines a cursor glyph. The *index* parameter is an index into the table of defined cursors, and the *cursor* parameter is an array of bits of the correct size, which depends 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 of the cursor bitmap 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 bitmap 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. The syntax is as follows:

```
void defcursor(Int32 index, Uint16 *cursor)
```

curorigin Subroutine

The **curorigin** sets the origin of a cursor. The origin is the point on the cursor that aligns with the current cursor valuators. The lower-left corner of the cursor has coordinates (0,0). Before calling the **curorigin** subroutine, you must define the cursor with the **defcursor** subroutine. The *index* parameter is an index into the cursor table created by the **defcursor** subroutine. The **curorigin** subroutine does not take effect until there is a call to the **setcursor** subroutine. The syntax is as follows:

```
void curorigin(Int16 index, Int16 xorigin, Int16 yorigin)
```

setcursor Subroutine

The **setcursor** subroutine sets the cursor characteristics. It selects a cursor glyph from among those defined with the **defcursor** subroutine. The *index* parameter picks a glyph from the definition table. The *color* and *writemask* parameters are ignored. They are present for compatibility with older systems that still make use of them. Set the color for the cursor with the **mapcolor** and **drawmode** subroutines. The syntax is as follows:

```
void setcursor(Int16 index,
               Colorindex color, Colorindex writemask)
```

getcursor Subroutine

The **getcursor** subroutine returns the cursor characteristics. It returns two values: the cursor glyph (in the *index* parameter) and a Boolean value (in the *bool* parameter) indicating whether the cursor is visible.

> Note: The *color* and *writemask* parameters are included for compatibility with previous versions; otherwise, they provide no useful information for current usage.

The syntax is as follows:

```
void getcursor(Int16 *index, Colorindex *color,
               Colorindex *writemask, Int32 *bool)
```

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.

The following example program defines a three-color, 32-by-32-bit cursor in the shape of an American flag with 12 stars:

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

main ()
{
    winopen("flag");
    setflag();
```

```
    color(0);
    clear();
    sleep(20);
}
setflag()
{
    static short        curs2[128] = {
        0,      0,      0,      0,
        0,      0,      0,      0,
        0,      0,      0,      0,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff, 0xffff, 0xffff, 0xffff,
     0,     0xffff, 0x6666, 0xffff,
    0x6666, 0xffff,      0, 0xffff,
     0,     0xffff, 0x6666, 0xffff,
    0x6666, 0xffff,      0, 0xffff,
     0,     0xffff, 0x6666, 0xffff,
    0x6666, 0xffff,      0, 0xffff,
     0,      0,      0,      0,
     0,      0,      0,      0,
     0,      0,      0,      0,
     0,      0,      0,      0,
    0xffff, 0xffff, 0xffff, 0xffff,
     0,      0,      0,      0,
    0xffff, 0xffff, 0xffff, 0xffff,
     0,      0,      0,      0,
    0xffff, 0xffff, 0xffff, 0xffff,
     0,      0,      0,      0,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff,      0, 0xffff,      0,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff,      0, 0xffff,      0,
    0xffff, 0xffff, 0xffff, 0xffff,
    0xffff,      0, 0xffff,      0 };
    curstype(C32X2);
    drawmode(CURSORDRAW);
    mapcolor(1,255,0,0);
    mapcolor(2,0,0,255);
    mapcolor(3,255,255,255);
    defcursor(1,curs2);
    setcursor(1,0,0);
    drawmode(NORMALDRAW);
}
```

Using the Keyboard

The keyboard class of special devices reports character values when keys, or a combination of keys, are pressed. The following section discusses both international keyboard input and controlling keyboard characteristics and behavior.

List of GL Keyboard Subroutines

clkon,clkoff	Turns keyboard click on and off.
lampon,lampoff	Turns the keyboard display lights on and off.
ringbell	Rings the keyboard bell.
setbell	Sets the duration of the keyboard bell sound.

setdblights	Sets the lights on the dial and switch box.

International Keyboard Input

The KEYBD device returns code points that correspond to the keys typed on the keyboard. The manner in which keystrokes are matched with the returned value depends on the *locale*, a language, territory, and code set combination used to identify a set of language conventions. For example, in United States-English installation, the value returned is the ASCII value that corresponds to the key pressed, taking into account the Shift and Ctrl keys. In other language keyboard installations, encodings from the ISO8859 family of code sets are returned. For further information on code sets, see Code Sets Overview in in *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*..

Input Method

The method used for matching up keystrokes to returned values is called the *input method*. The locale, which determines the key mapping, is determined from the default setting when the system was configured, or from the $LANG environment variable, or from the most recent invocation of the **setlocale** subroutine. The input method helps correct for different key placements on different keyboards; for instance, the letter Z on the French keyboard appears in the same location as the letter W on the US English keyboard. For further information on input method, see Input Method Overview in in *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*.

The input method also automatically takes into account cases where multiple keystrokes are needed to specify one unique letter. The value returned may be a single-byte value (in the case of languages with single-byte code sets), or double-byte value (for languages supporting double-byte code sets). For further information on National Language Support, refer to the National Language Support Overview for Programming in in *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*.

In the example of the United States-English keyboard device, events are reported only on a key downstroke. The value returned takes into account the state of the modifier key (the Ctrl, Shift, and Shift-lock keys). Pressing the lowercase letter a on the keyboard returns the ASCII value 0x61, while pressing Shift-A returns the ASCII value 0x41. Similarly, pressing the Ctrl-G key sequence returns BEL (hex value 0x07), the Ctrl-D key sequence returns ETX (hex value 0x03) and so on, proceeding with the standard ASCII mapping.

It is important to understand the difference between the device and the values it returns when you queue the keyboard. If your program contains the following instruction:

```
qdevice(KEYBD);
```

then the statement:

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

returns the following:

```
dev == KEYBD
val == the ISO8859 code set encoding of the character pressed.
```

In United States-English keyboard installations, you can test for individual keystrokes by using instructions with the following format:

```
qdevice(AKEY);
```

This returns the AKEY device when the uppercase A or lowercase a key is pressed.

> Note: Currently, the keyboard mapping for KEY devices is guaranteed to be correct only in United States-English installations.

Keyboard Mapping

The actual keyboard mapping used by the KEYBD device is contained in an **imkeymap** file. The following locations are searched, in the order indicated, for this file:

1. **$XDIR/imkeymap**
2. **$HOME/imkeymap**
3. **/usr/lib/nls/loc/$LANG**

If none of these files are found, the mapping defaults to the ISO8859 (Latin-1) keyboard mapping. An input method determines the mapping. By default, the input method used is IMSimpleMap. For most country and language combinations, two different encodings are supplied: the ISO8859 encoding and the IBM-850 encoding. For example, $LANG = Fr_CH is used to indicate the IBM-850 encoding for Swiss French, while $LANG = fr_FR indicates that the ISO8859-1 encoding should be used.

> Note: Changing the locale after initialization has no effect. Keyboard initialization occurs at the time the X server is brought up. Therefore, the locale (or $LANG variable) should be set before the X server is started.

For applications developed for use in countries and languages with more complex input-composing requirements, such as in the Asian/Pacific Rim regions, it is strongly suggested that the AIXwindows input widget be used. Use of AIXwindows for input requires knowledge of the X display connection and the X Window identifier of a GL window for a GL session. This information can be obtained with the GL **getXdpy** or **getXwid** subroutines. For a review of limitations, see Using Enhanced X-Windows Calls with GL Subroutines. Example usages can be found in the **/usr/lpp/GL/examples** directory.

Alt PF Key Keystroke Sequences

In the normal course of operations, the window manager interprets some key sequences as having special meaning. These key sequences typically involve pressing a PF key while holding down the Alt key, and are used to modify a window by moving, resizing, raising, lowering it, and so on. The window manager removes these events from the event queue, and a GL application requesting these key events does not receive them. The actual keystrokes that are grabbed are under control of the **.mwmrc** file, and are known as *accelerators*. Although the default accelerators shipped with the system are Alt-PF key sequences, the user can change or eliminate the key sequences used as menu accelerators by editing the **$HOME/.mwmrc** file. Note, however, that changes made to the **$HOME/.mwmrc** file affect all applications. Currently, there is no selective means of disabling the key grabbing for some, but not all, applications.

Controlling the Keyboard

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

clkon and clkoff Subroutines

The **clkon** and **clkoff** subroutines turn the keyboard click on and off. The syntax for the **clkon** and **clkoff** subroutines is as follows:

```
void clkon()
void clkoff()
```

lampon and lampoff Subroutines

The **lampon** and **lampoff** subroutines control the four lamps on the keyboard. Each 1 (as opposed to 0) in the four lower-order bits of the *lamps* parameter to the **lampon** subroutine causes the corresponding keyboard lamp to be on or off, depending on the subroutine called. The syntax for the **lampon** and **lampoff** subroutines is as follows:

```
void lampon(Int8 lamps)
void lampoff(Int8 lamps)
```

ringbell Subroutine

The **ringbell** subroutine rings the keyboard bell. The syntax is as follows:

```
void ringbell()
```

setbell Subroutine

The **setbell** subroutine sets the duration of the keyboard bell. A value of 0 in the *durat* parameter is off, 1 is a short beep, and 2 is a long beep. The syntax is as follows:

```
void setbell(Char8 durat)
```

setdblights Subroutine

The **setdblights** subroutine 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 of the mask must be set to `1 (1<<3)|(1<<7)=0x88`. The syntax is as follows:

```
void setdblights(Int32 mask)
```

Controlling Queues and Devices

This section discusses the following areas:

- Polling and Queuing
- Polling a Device
- Event Queue
- Input Focus
- Special Devices
- Controlling Peripheral Input/Output Devices

GL supports three classes of input devices:

valuators	Return an integer value. For example, a dial on a dial and button box is a valuator. A mouse is a pair of valuators: one reports horizontal position and the other reports vertical position.
buttons	Return a Boolean value: FALSE when they are not pressed (open) and TRUE when they are pressed (closed). Examples of buttons are keys on an unencoded keyboard, buttons on a mouse, and the switches on a dial and button box.
other devices	Return information about other system events. For example, the keyboard returns ASCII characters. Most of these special 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. If you hold down the Shift key and then press the A key, an ASCII A is reported.

List of GL Queue and Device Control Subroutines

blkqread	Reads multiple entries from the event queue.
getbutton	Returns the current state of a button.
getdev	Reads a list of valuators.
getvaluator	Returns the current state of a valuator.
isqueued	Indicates whether a specified device is enabled for event queuing.
noise	Filters valuator motion.
qdevice	Enables an input device for event queuing.
qenter	Creates an event queue entry.
qread	Reads the first entry in the event queue.

qreset	Empties the event queue.
qtest	Checks the contents of the event queue.
setvaluator	Assigns an initial value to a valuator.
tie	Ties two valuators to a button.
unqdevice	Disables an input device for event queuing.

Understanding Windows and Input Control in GL introduces window and input control categories.

Polling and Queuing

Most input devices have an associated current value. If the input device is a button, the value is either 1 (pressed) or 0 (not pressed). If the device is a valuator, such as a dial or the *x* position of the mouse, its value is an integer that indicates the position of the device. Some special devices do not have an associated current value.

A program can get the value from input devices by either polling or queuing:

- Polling immediately returns the value of a device that is a button or valuator. For example, a `getbutton(LEFTMOUSE)` subroutine call returns 1 if the left button of the mouse is down, and returns 0 if it is up.
- Queuing uses an event queue to save changes in device values and other input events so the program can read them later.

Input Queue versus Polling

In most cases, using the input queue is better than polling. For example, if you press and release a mouse button in an instant, a program that uses polling can miss the event (that the button was down) if it happened between two calls to `getbutton(LEFTMOUSE)`.

For example, in a drawing program where you may want to indicate a series of vertices quickly and the system's calculations cannot keep up, queuing saves all the state changes so nothing is missed. This is so even if the program is doing something else when the event happens.

Devices that are queued act as asynchronous devices, independent of the user process. Whenever a device that is queued changes state, an entry is made in the event queue. If a program reads the queue in a timely fashion, no events are lost.

You can decide which devices, if any, to queue and establish some rules about what constitutes a state change, or event, for those devices. By default, no devices are queued.

Another difference between polling and queuing is that as long as a polled I/O device is active, it continues to send its value to the program that polls it. For instance, a call to `getbutton(LEFTMOUSE)` continues returning a value as long as you hold down the mouse button. In contrast, if the left mouse button had been queued, you could hold it down for any length of time and only one event is added to the queue. A second event is added to the queue only when the button is released and pressed a second time.

In addition to input subroutines, other subroutines control the characteristics of the peripheral input/output devices (see Using the Keyboard). These subroutines turn the keyboard click and the keyboard lights on and off; ring the keyboard bell; and control the lights on the dial and button box.

Device Descriptions

See the Input Buttons, Input Valuators, and Special Devices tables for listings of specific devices and their descriptions. Special devices, such as window manager, cursor, and ghost devices, are discussed in Special Devices.

Input Buttons

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, BKEY...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

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

Special Devices	
Device	**Description**
QFULL	Creates event when device queue overflows

Polling a Device

You can poll a device to determine its current state.

getbutton Subroutine

The **getbutton** subroutine polls a button and returns its current state. The *number* parameter specifies the number of the device you want to poll. The **getbutton** subroutine returns either TRUE or FALSE. TRUE indicates the button is pressed. The syntax is as follows:

```
Int32 getbutton(Device number)
```

getdev Subroutine

The **getdev** subroutine polls up to 128 valuators and buttons at one time. You specify the number of devices you want to poll and an array of devices. (See the Input Buttons and Input Valuators tables for listings of devices.) The *values* parameter returns the state of each device in the corresponding array location. The syntax is as follows:

```
void getdev(Int32 number, Device *devices, Int16 *values)
```

getvaluator Subroutine

To determine the values of valuators, use the **getvaluator** subroutine. You can poll any valuator, whether it is queued or not. The *device* parameter specifies a valuator device number, whose value reflects the current state of the device. The syntax is as follows:

```
Int32 getvaluator(Device device)
```

Event Queue

Input devices can make entries in the event queue. Each entry includes the device number and a device value. The **qdevice** subroutine enables queuing of events from an input device. The **unqdevice** subroutine indicates that a device is no longer queued. The **isqueued** subroutine tells you if a specific device is queued. The three subroutines most commonly used for queuing are **qdevice**, **qread**, and **qtest**.

The input queue can contain up to 101 events at a time. To check for overflow, you can queue the QFULL device. 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 the **qread** subroutine at the point in the input queue at which data was lost.

qdevice Subroutine

The **qdevice** subroutine queues the specified device (a keyboard, button, or valuator). The *device* parameter specifies a device name. Each time the device changes state, an entry is made in the event queue. The syntax is as follows:

```
void qdevice(Device device)
```

qtest Subroutine

The **qtest** subroutine returns the device number of the first entry in the event queue; if the queue is empty, the subroutine returns zero. The **qtest** subroutine always returns immediately to the caller and makes no changes to the queue. The syntax is as follows:

```
Int32 qtest()
```

qread Subroutine

The **qread** subroutine, like the **qtest** subroutine, returns the device number of the first entry in the event queue. However, if the queue is empty, the subroutine waits until an event is added to the queue. The **qread** subroutine returns the device number, writes the data part of the entry into data, and removes the entry from the queue. The syntax is as follows:

```
Int32 qread(Int16 data)
```

qreset Subroutine

The **qreset** subroutine removes all entries from the event queue and discards them. The syntax is as follows:

```
void qreset()
```

Polling and Queuing Example Program

The following C language code uses both polling and queuing to control a simple paint program. To determine the color, the program reads characters typed from the keyboard. To determine the drawing location, the program polls the LEFTMOUSE device and draws a 5-pixel-wide circle if the value for the device is TRUE.

```
while (TRUE) {
 while (qtest() || !attached) {
  dev=qread(&value);
```

```
   if (dev == ESCKEY) {
    exit(0);
   }
   else if (dev == REDRAW) {
   /* if first in queue */
    reshapeviewport();
    color (BLUE);
    clear();
   }
   else if (dev == RKEY) {
    color(RED);
   }
   else if (dev == GKEY) {
    color(GREEN);
   }
   else if (dev == BKEY) {
    color (BLUE);
   }
   else if (dev == INPUTCHANGE) attached = value;
  }  /*end of while qtest or not attached */
  mdraw();
 } /* end of while (TRUE) */
} /* end of main() */
mdraw() {
 int ix;
 int iy;
 if (getbutton(LEFTMOUSE)) {
  ix=getvaluator(MOUSEX);
  iy=getvaluator(MOUSEY);
  circfi (ix, iy, 5);
 }
}
```

This example assumes that you have initialized the program appropriately, especially regarding the use of the **qdevice** subroutine to enable the R, G, and B keys for queuing.

tie Subroutine

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. The **tie** subroutine takes three parameters: *button* and two valuators, *val1* and *val2*. The syntax is as follows:

```
void tie(Device button, Device val1, Device val2)
```

Whenever the button changes state, three entries are made in the queue that record the current state of the button and the current position of each valuator. You can tie one valuator to a button by making the *val2* parameter equal to zero. You can untie a button from valuators by making both the *val1* and *val2* parameters equal to zero.

noise Subroutine

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

```
void noise(Device valuator, Int16 delta)
```

qenter Subroutine

The **qenter** subroutine creates event queue entries. It places entries directly into the event queue. The **qenter** subroutine takes two 16-bit integers, given in the *qtype* and *value* parameters, and enters them into the event queue. The syntax is as follows:

```
void qenter(Int16 qtype, Int16 value)
```

unqdevice Subroutine

Use the **unqdevice** subroutine to disable the queuing of events from a device. If the device has recorded events in the queue that have not been read, those events remain in the queue. (You can use the **qreset** subroutine to flush the event queue.) The syntax is as follows:

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

isqueued Subroutine

The **isqueued** subroutine indicates whether a specific device is queued. It returns a Boolean value. A value of TRUE indicates that the device is enabled for queuing; FALSE indicates that the device is not queued. The syntax is as follows:

```
Int32 isqueued(Int16 device)
```

blkqread Subroutine

The **blkqread** subroutine returns multiple queue entries. Its first parameter, *data*, specifies an array of short integers, and its second parameter, *number*, specifies the size of the array data. The **blkqread** subroutine returns the number of 16-bit integers stored in the array specified in the *data* parameter. The array is filled alternately with device numbers and device values. Because each queue entry consists of two 16-bit words, the number of entries read is twice the number of queue entries and cannot be more than the value of the *number* parameter. The syntax is as follows:

```
Int32 blkqread(Int16 *data, Int16 number)
```

You can also use this subroutine 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 significant.

Input Focus

Keyboard, mouse, and valuator events are added to the event queue only when the application window has input focus. Input focus is controlled by the window manager. Windows with focus are usually highlighted in some fashion (typically by changing the border color of the window).

> Note: In a multiwindow application, only one of the application windows can have input focus at a time.

The default window manager for your system is the AIXwindows window manager. The behavior of this window manager can be controlled by appropriately editing the **./.Xdefaults** file in the user's home directory. The focus policy is controlled by the **keyboardFocusPolicy** resource. If the following line is added to the **.Xdefaults** file:

```
Mwm*keyboardFocusPolicy: pointer
```

the window with the cursor in it has input focus. Alternatively, if the following line is added to the **.Xdefaults** file:

```
Mwm*keyboardFocusPolicy: explicit
```

the user must click with the left mouse button inside a window to give that window input focus.

Notes:

1. The window focus affects how the MOUSEX and MOUSEY events are reported. In particular, if `pointer` focus is selected, mouse events are reported only when the cursor is inside one of the application's windows. If `explicit` focus is selected, mouse events may be reported, depending on which window has the focus, even if the cursor is outside any of the application's windows.
2. The focus policy determines how INPUTCHANGE events are queued. An INPUTCHANGE event is queued when an application's window receives or loses input focus. If `pointer` focus is selected, an INPUTCHANGE event is queued when the cursor enters or leaves an application window. If `explicit` focus is selected, the INPUTCHANGE token is generated only when the user makes a focus change by clicking the mouse elsewhere.

Special Devices

This section discusses the five types of special devices:

- Keyboard
- Timer
- Cursor
- Ghost
- Window manager

Keyboard Devices

(See Using the Keyboard.)

Timer Devices

The timer devices record an event every 60th of a second. You can use a timer device to synchronize a graphics program with a real clock. To record events less frequently, use the **noise** subroutine. For example, if you call:

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

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

Cursor Devices

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

Ghost Devices

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

Window Manager Devices

The following devices can be queued by the user application to obtain window manager events. Some of these devices are queued automatically when a window is opened. Some have to be queued explicitly. All of these devices return the window ID of the window associated with the event. See Creating and Managing Windows for more information.

REDRAW

The window manager inserts a redraw token each time a window becomes exposed and needs to be redrawn. The REDRAW device is queued automatically when a window is opened.

REDRAW events are generated for a window whenever the following events occur:

1. the window is uncovered because another window has been moved away or pushed below it.
2. the window has been resized smaller or larger by the user.
3. whenever the window is moved.

In the current implementation, the contents of the z-buffer are not copied to the new location on a window move; therefore, a REDRAW event is generated out of necessity. In the current implementation, REDRAW events are *not* generated if the overlay portion of a window has been drawn into or otherwise affected by other windows, other GL applications, or the use of the **fullscrn**, **endfullscrn** subroutines. Currently, the REDRAWOVERLAY pseudodevice is not supported.

REDRAWICONIC

The window manager sends this token when a window needs to be redrawn as an icon. The REDRAWICONIC device is queued automatically when the **iconsize** subroutine is called.

DEPTHCHANGE	When queued, this device indicates an open window has been pushed or popped. The value of the token is the `gid` of the window that has changed. Use the **windepth** subroutine to determine the stacking order. **Note:** The DEPTHCHANGE device is currently tied to the INPUTCHANGE device; DEPTHCHANGE events are only generated when INPUTCHANGE events are. DEPTHCHANGE events are reocrded only relative to the current process; changes of window stacking order involving other GL applications and/or X clients are not recorded.
WINSHUT	When queued, the window manager sends this token when the `Close` item is selected from the window manager's title bar option menu. If the WINSHUT device is not queued, the `Close` item on the program's window menu appears grayed-out and has no effect if selected. Do not confuse the WINSHUT with the WINQUIT device. The WINSHUT device is used by the applications program to shut a window; the WINQUIT device is used to quit and exit the program. **Note:** The WINSHUT device is not currently supported.
WINQUIT	When queued, the window manager sends this token rather than killing a process when the `Quit` item is selected from the window manager's title bar option menu. If this device is not quit, the window manager issues a `kill -15` command to the process ID of the process owning the window.
WINFREEZE	If queued, the window manager sends this token when a window is stowed to icons, rather than blocking the processes of the stowed windows. This device should be queued if the program is designed to draw an icon (see the **iconsize** subroutine) or is a multiwindow application. In other words, if one window of a multiwindow application is stowed, this device allows the owner process to continue.
WINTHAW	If queued, the window manager sends this token when a window is unstowed.
INPUTCHANGE	The window manager inserts an INPUTCHANGE token when a window is given input focus. The value inserted with the token is the window ID of the window receiving focus. A value of 0 (zero) is returned if the window given focus belongs to another application. The INPUTCHANGE device is queued automatically when a window is opened.
PIECECHANGE	If queued, this device indicates that a window has been exposed because another window has been moved away. This token is *not* sent if the window has been unstowed from an icon, or has been exposed due to a depth change.

Controlling Peripheral Input/Output Devices

The application programmer can set the initial value of a valuator device with the **setvaluator** subroutine.

setvaluator Subroutine

Valuators are single-value input devices: for example, the horizontal and vertical motion of mouse, or the turning of a dial. The value is a 16-bit integer. The **setvaluator** subroutine assigns an initial value (the *init* parameter) to a valuator. The *min* and *max* parameters specify the minimum and maximum values the device can assume. The syntax is as follows:

```
void setvaluator(Device val, Int16 init, Int16 min, Int16 max)
```

In addition to subroutines that poll and queue input devices, there are those that control the characteristics and behavior of the GL peripheral input/output devices. See Using the Keyboard for this information.

Querying the System

Certain GL subroutines obtain information about various processes in or aspects of the system. These include current cursor characteristics, maximum character height in the current raster font, and the current display, color map, or drawing mode.

Other subroutines furnish lists, such as the **blkqread** subroutine, which reads a block of event queue entries, and the **getdev** subroutine, which polls the specified valuators.

List of GL Query Subroutines

blkqread	Reads multiple entries from the event queue.
genobj	Returns a unique integer for use as an object identifier.
gentag	Returns a unique integer for use as a tag number.
getbackface	Indicates whether backfacing polygon removal is on or off.
getbuffer	Indicates which buffers are enabled for drawing.
getbutton	Returns the current state of a button.
getcmmode	Returns the organization of the current color map.
getcolor	Returns the current color in color map mode.
getcpos	Returns the current character position.
getcursor	Returns the cursor characteristics.
getdcm	Indicates whether depth-cue mode is on or off.
getdescender	Returns the baseline extent of the longest character descender.
getdev	Reads a list of valuators.
getdisplaymode	Returns the current display mode.
getdrawmode	Returns the current drawing mode.
getfont	Returns the current raster font number.
getgdesc	Returns information about currently installed graphics hardware.
getgpos	Gets the current graphics position.
getheight	Returns the maximum character height in the current raster font.
getlsrepeat	Returns the linestyle repeat count.
getlstyle	Returns the current linestyle.
getlwidth	Returns the current linewidth.
getmap	Returns the number of the current color map.
getmatrix	Gets a copy of the current transformation matrix.
getmcolor	Gets a copy of the RGB values for a color map entry.
getmcolors	Returns a range of color map RGB values.
getmmode	Returns the current matrix mode.
getnurbsproperty	Returns the current value of a trimmed NURBS surfaces display property.
getopenobj	Returns the current open object.
getorigin	Returns the position of a window.
getpattern	Returns the index of the current fill pattern.

getplanes	Returns the number of available bitplanes.
getscrmask	Returns the current screenmask.
getsize	Returns the size of a window.
getsm	Returns the current shading style used to draw filled polygons.
getvaluator	Returns the current state of a valuator.
getviewport	Gets a copy of the dimensions of the current viewport.
getwritemask	Returns the current writemask.
getzbuffer	Indicates whether z-buffering is on or off.
gRGBcolor	Returns the current color in RGB mode.
gRGBmask	Returns the current RGB writemask.
gversion	Returns the version of GL being used.
qread	Reads the first entry in the event queue.
strwidth	Returns the width of the specified text string.
winget	Returns the identifier of the current window.

Creating and Managing Pop-Up Menus

This section discusses the following topics:

- Creating a Menu
- Calling Up a Pop-Up Menu
- Advanced Menu Formats

List of GL Pop-Up Menu Subroutines

addtopup	Adds an item to an existing pop-up menu.
defpup	Defines a pop-up menu.
dopup	Displays a pop-up menu.
freepup	Frees (deallocates) a pop-up menu and its data structures.
newpup	Allocates and initializes the structure for a new pop-up menu.
setpup	Enables or disables a given pop-up entry.

Note: Using popup menu facilities when the X server has been initialized in layer 1 (the overlay bitplane) is ugly. The GL popup menus use the **fullscrn** subroutine.

Creating a Menu

GL contains subroutines that you can include in your graphics programs to create and use pop-up menus. When you select an item from a menu, these subroutines automatically identify which menu item has been selected.

To create a pop-up menu in C programming language, use the **defpup subroutine**. To create a pop-up menu in FORTRAN, use the **newpup** and **addtopup** subroutine.

You can also design your own pop-up menu interface using the overlay bitplanes.

To set pop-up menu colors, use the **mapcolor** and **mapcolors** subroutines. To facilitate color selection, the following tokens are defined in the **/usr/include/gl/gl.h** file:

- PUP_CLEAR
- PUP_COLOR
- PUP_BLACK
- PUP_WHITE

defpup Subroutine

The **defpup** subroutine defines a pop-up menu by allocating structure and making menu entries. In C programming language, you can combine the functions of the **newpup** and **addtopup** subroutines by calling the **defpup** subroutine, available only in C programming language. FORTRAN does not support variable parameter lists. The syntax is as follows:

```
Int32 defpup(Char8 *String [, Int32 arguments ...])
```

The **defpup** subroutine creates the menu in one step. You can add menu entries with the **addtopup** subroutine.

The PUPDRAW option of the **drawmode** subroutine allows you to define the colors for pop-up menus with the **mapcolor** subroutine. Be aware that you cannot draw while using the PUPDRAW option of the **drawmode** subroutine.

If you program in FORTRAN, you must use the **newpup** and **addtopup** subroutines to create the menu.

newpup Subroutine

The **newpup** subroutine allocates and initializes a structure for a new menu. This subroutine takes no arguments and returns a 32-bit integer identifier (the *popup* parameter) for the pop-up menu. The syntax is as follows:

```
Int32 newpup()
```

addtopup Subroutine

After the **newpup** subroutine creates an empty menu, use the **addtopup** subroutine to build the menu by adding entries to the bottom of the empty menu structure. The syntax is as follows:

```
void addtopup(Int32 popup, Char8 *String, Int32 argument)
```

The *popup* parameter is the menu identifier returned by the **newpup** subroutine or the **defpup** subroutine. The *String* parameter is a character string that specifies the entries in the menu. The string lists the menu labels from the top to the bottom of the menu, with a | (vertical bar delimiter) between entries.

The FORTRAN version, the **addtopup** subroutine, takes an additional *Length* parameter, which specifies the number of characters in the string. The *Argument* parameter is necessary only for advanced menu formats.

Following are two examples of adding to a menu:

In C programming language:

```
menu = newpup();
addtopup (menu, "first|second|third");
```

In FORTRAN:

```
IMENU = NEWPUP();
CALL ADDTOP (IMENU, "first|second|third", 18);
```

The number 18 in the FORTRAN subroutine is the number of characters in the string, including the vertical bar delimiters.

Calling Up a Pop-Up Menu

The following subroutines allow you to invoke your choice of pop-up menus and free the memory occupied by the menu when no longer needed.

> Note: Do not call the exit routine directly from a pop-up menu. This could cause the pop-up menu to be erased from the overlays. To prevent this, either return from the **dopup** subroutine or clear the overlays before you call the exit routine.

dopup Subroutine

The **dopup** subroutine displays a pop-up menu previously defined with the **defpup** subroutine. The *popup* parameter is the identifier of the pop-up menu. The system displays the menu until you either make a selection or release the button with the cursor off the menu. The value that the **dopup** subroutine returns depends on the menu selection. If no selection is made, the **dopup** subroutine returns -1 (negative one). If the pop-up menu does not use an advanced menu format, the **dopup** subroutine returns an integer corresponding to the position of the item in the menu. The syntax is as follows:

```
Int32 dopup(Int32 popup)
```

To cause the right mouse button to bring up the menu built in the previous example, use the following code:

In C programming language:

```
dev = qread(&val);
if (dev == RIGHTMOUSE) {
    if (val == 1) {      /* right mouse button is pressed */
        menuval = dopup (menu);
    }
}
```

In FORTRAN:

```
IDEV = QREAD(IVAL)
IF (IDEV .EQ. RIGHTM) THEN
    IF (IVAL .EQ. 1) THEN
        MENVAL = DOPUP (IMENU);
    ENDIF
ENDIF
```

Select the `first`, `second`, or `third` item in the menu by positioning the cursor over one of these items, then releasing the button. To make no selection, release the button with the cursor off the menu. The following table shows the return value for each possible selection in the example:

Default Return Values	
Selection	**Return Value**
First	1
Second	2
Third	3
No selection	-1

freepup Subroutine

The **freepup** subroutine deletes a pop-up menu, freeing the memory reserved for its data structures. The syntax is as follows:

```
void freepup(Int32 popup)
```

setpup Subroutine

The **setpup** subroutine enables or disables a given pop-up menu entry. If used properly, this subroutine renders submenus associated with a disabled entry inaccessible. The syntax is as follows:

```
void setpup(Int32 pup, Int 32 entry, Int32 mode)
```

Advanced Menu Formats

You can use advanced menu format to:

- Change the value returned when you select a menu item.
- Bind a function to a whole menu or to a menu item.
- Make a title bar or a nested (rollover) menu.

You introduce the special format when you call the **defpup** or **addtopup** subroutine. Following the string that defines the menu entry, add these format instructions:

- Percent sign (%) begins the special format.
- One format character specifies which format to use (see the Summary of Advanced Menu Formats table).
- Numeric values, parameters, or both, are required for some formats. Numeric values immediately follow the format character. Parameters always come at the end of the **defpup** or **addtopup** subroutine.

More than one special format can be associated with a menu entry. Use a | (vertical bar) to separate entries.

Summary of Advanced Menu Formats				
Task	**Format**	**Changes Return Value?**	**Needs Numeric Value?**	**Needs Parameters?**
Return default values	%n	No	No	No
Return other values	%x	Yes	Yes	No
Make title	%t	No	No	No
Bind function/ whole menu	%F	Yes	Possibly	Yes
Bind function/ menu item	%f	Yes	Possibly	Yes
Make nested menu	%m	Yes	No	Yes

Returning the Default Values for Menu Selections

The %n format can be used to return a menu entry to its default settings. The %n format takes no parameters. The following menu:

```
menu = newpup();
addtopup(menu, "first|second|third");
```

is the same as this menu:

```
menu = newpup();
addtopup(menu, "first %n|second %n|third %n");
```

Changing the Return Values for Menu Selections

The %x format changes the numeric value that the **dopup** subroutine returns. Note the following menu:

```
menu = newpup();
addtopup(menu, "first %x15|second %x7");
```

Selecting `first` causes the **dopup** subroutine to return 15, not -1 (the default). Selecting `second` causes the **dopup** subroutine to return 7, not 2 (the default). For this format, you must specify a numeric value that the **dopup** subroutine returns in place of the default.

Making a Title Bar

The %t format creates a title bar on a pop-up menu. You cannot select the title bar. It does not highlight.

The following menu is the same as the first example, except that there is a title bar at the top of the menu. The %t format takes no parameters.

```
menu = newpup();
addtopup(menu, "Cardinal %t|first|second|third");
```

Binding a Function to a Whole Menu

The %F format specifies a function that affects all values returned by any item in the menu. This format requires a parameter to specify the function or subroutine that affects all values that the **dopup** subroutine returns.

```
menu = newpup();
addtopup(menu, "Cardinal %t %F|first|second %x10", funct);
```

When the user selects `first` from the pop-up menu, the **dopup** subroutine returns `funct(1)` instead of 1. When the user selects `second`, the **dopup** subroutine returns `funct(10)`.

Binding a Function to a Menu Entry

The %f format makes a menu entry that calls a function or subroutine. The name of the function is the parameter to this format:

```
menu = newpup();
addtopup(menu, "first|call %f", func);
```

When the user selects `call` from the pop-up menu, the `funct` function is called with a parameter of 2 (the default return value for this selection). The `dopup(menu)` returns `funct(2)`.

Call the **addtopup** subroutine each time you want to add another menu entry that has its own function.

You can also use the **defpup** subroutine to define a pop-up with one or more menu entries that bind to a function.

Making a Nested (Rollover) Menu

The %m format creates a simple nested pop-up menu (a submenu). When you roll the cursor to the right side of the menu item, you invoke the submenu. Labels on the submenu can have further choices. This format requires a parameter to specify the menu identifier for the pop-up submenu.

Note: You must declare any submenus before the main menu in your program.

```
submenu = newpup();
addtopup(submenu, "one|two");
menu = newpup();
addtopup(menu, "Cardinal %t|above %x5|below %m", submenu);
```

If you select an item from the submenu, the `dopup(menu)` returns the same value as the `dopup(submenu)`. If you display the submenu without making a selection, the **dopup** subroutine returns negative 1.

Working with the Textport

GL provides subroutines both to set up a textport, a window associated with a graphics application shell, and to turn it on and off. The **textport** subroutine allocates an area of the screen for a textport, and the parameters specify the screen coordinates for the textport. The syntax is as follows:

```
void textport(Screencoord left, Screencoord right,
              Screencoord bottom, Screencoord top
```

The **tpon** subroutine brings the textport to the front of any windows that conceal it so that character strings can be drawn into it. The **tpoff** subroutine pushes the textport behind all other windows. When the textport is off, it is not visible on the screen, and character strings cannot be written into it. The syntax for the **tpon** and **tpoff** subroutines is as follows:

```
void tpon()
void tpoff()
```

List of GL Textport Subroutines

textport	Allocates an area of the screen for a textport.
tpoff	Turns off the textport.
tpon	Turns on the textport.

Chapter 13. Using Enhanced X-Windows Calls with GL Subroutines

This section describes how GL drawing subroutines can be used with AIXwindows window management routines to control the mapping of windows, set window properties, and create widgets. The subroutines and techniques described in this section allow the user to mix Enhanced X-Windows and GL calls within the same program. However, not all possible mixings of Enhanced X-Windows and GL within the same program result in predictable behavior. To use GL and Enhanced X-Windows together constructively, it is important to have a basic understanding of the internal structure and implementation of GL in the AIXwindows Environment/6000.

In general, there are two kinds of GL subroutines: those used for drawing or controlling the hardware and those used for window management, obtaining input, and managing pop-up menus. Drawing subroutines (such as **v3f**, **loadmatrix**, and **nurbssurface**) and hardware control subroutines (such as **gconfig** and **zfunction**) access and control the graphics adapter directly. That is, there are no complex code layers between these subroutines and the graphics adapter. Specifically, this means that the X server remains ignorant of the current state and contents of a GL window. Eliminating complex code layers improves calculation time.

In contrast, window management subroutines (such as **winpop** and **icontitle**) and input subroutines (such as **qread** and **getbutton**) are implemented as a layer on top of the Enhanced X-Windows Protocol. As a result, all input and events are obtained from the X server, and the X server knows about and carries out all requests to create, move, map, and resize windows. In this way, a single centralized process manages all windows visible on the screen.

The differences between direct access subroutines and those that work through the X server affect how GL and Enhanced X-Windows can and cannot be used together. The following section discusses restrictions on GL and Enhanced X-Windows use. Later sections discuss and give examples of how GL and Enhanced X-Windows calls can be used together.

For more information on using Enhanced X-Windows calls with GL subroutines, see:

- Restrictions on Using Enhanced X-Windows Calls with GL Subroutines
- Enhanced X-Windows and GL Interoperability

> Note: This section deals with an advanced topic and may be difficult to understand unless the reader has a good conceptual and practical grasp of GL and the Enhanced X-Windows System.

Restrictions on Using Enhanced X-Windows Calls with GL Subroutines

Some GL and Enhanced X-Windows calls cannot be used together. Only a few routines make sense and are valid, while all others result in undefined behavior.

As mentioned previously, the GL subroutine library is partitioned into two pieces: one, of which the X server remains ignorant, and the second, which operates through Enhanced X-Windows. Due to this partitioning, certain Enhanced X-Windows calls, when invoked with GL calls, result in unpredictable behavior. For this reason, do not use these calls together. The following sections outline restrictions on calls:

- Rendering Models
- Color Maps
- Fonts
- Internal Properties
- Fullscreen Mode

- Coordinate Transformation
- Mixed GL and Windows Input

List of GL Enhanced X-Windows Subroutines

getXdpy	Returns the Enhanced X-Windows connection for the given GL session.
getXwid	Returns the Enhanced X-Windows ID of the current GL window.
winX	Converts the specified Enhanced X-Windows window into a GL window.
finish	Blocks until all buffers and FIFOs are empty.

Rendering Models

Enhanced X-Windows calls should not be used to render into a GL window, and GL calls should not be used to render into an Enhanced X-Windows window. The X server is ignorant of the state of a GL window, and the GL subroutines are not aware of Enhanced X-Windows. If rendering into the other type of window is attempted, the most likely outcome is that anything drawn appears either in the wrong colors or the wrong place, or does not appear at all. In some situations, the application exits prematurely.

The previous restrictions do not imply that you cannot do Enhanced X-Windows and GL drawing within the same process, only that you should not use both rendering models to draw into the same *window*. As long as Enhanced X-Windows drawing is limited to Enhanced X-Windows windows, and GL drawing is limited to GL windows, there should be no conflicts.

The following is a partial list of Enhanced X-Windows and base operating system drawing routines. Do *not* use these routines to draw inside a GL window.

Enhanced X-Windows and base operating system Drawing Routines Restricted from GL Windows	
XClearArea	**XDrawString16**
XClearWindow	**XDrawText**
XDrawArc	**XDrawText16**
XDrawArcsXDrawRectangles	**XFillArc**
XDrawImageString	**XFillArcs**
XDrawImageString16	**XFillPolygon**
XDrawLine	**XFillRectangle**
XDrawLines	**XFillRectangles**
XDrawPoint	**XPutImage**
XDrawPoints	**XPutPixel**
XDrawRectangle	**XmStringDraw**
XDrawRectangles	**XmStringDrawImage**
XDrawSegments	**XmStringDrawUnderline**
XDrawString	

Do not use GL subroutines on an Enhanced X-Windows window unless that window has first been converted to a GL window with the **winX** subroutine.

Color Maps

The AIXwindows window manager does not always install a new colormap immediately. Usually, colormaps are installed only when the input focus changes. This problem can affect color index mode windows, as well as RGB mode windows on the POWERgraphics GTO, POWER Gt4, and POWER Gt4x adapters.

Although the X server performs overall color map management, including color map creation, allocation, and installation, do not use Enhanced X-Windows calls to change the GL color map, or GL calls to change X color maps.

Internally, GL maintains two color maps. One is the global color-index-mode color map, and it can be modified by any GL application. The other color map is an internal color map maintained for PseudoRGB graphics adapters. For 8-bit adapters, this internal palette is loaded with a 3-3-2 dithered color map. On 24-bit adapters, this palette is loaded with an 8-8-8 straight ramp. The actual format of either of these two color maps depends on which adapter is installed in your system. Modifying either of these color maps results in unpredictable colors on the screen.

The following is a partial list of Enhanced X-Windows color map manipulation routines. Do *not* use these routines to modify GL color maps.

Enhanced X-Windows Color Map Routines Restricted from GL	
XAllocColor	**XSetRGBColormaps**
XAllocColorCells	**XSetWindowColormaps**
XAllocColorPlanes	**XSetWMColormapWindows**
XAllocNamedColor	**XStoreColor**
XAllocStandardColormap	**XStoreColors**
XCopyColormapAndFree	**XStoreNamedColor**
XCreateColormap	**XUninstallColormap**
XInstallColormap	

Fonts

Do *not* use Enhanced X-Windows fonts calls to load a GL font. If you wish to use Enhanced X-Windows fonts with the GL **charstr** subroutine, use the GL **loadXfont** subroutine to access them. The following Enhanced X-Windows routines cannot be used to load a GL font and do not affect the current GL font:

- **XLoadFont**
- **XLoadQueryFont**

Likewise, do not attempt to use the GL **loadXfont** subroutine to get a font for Enhanced X-Windows rendering.

Although the internal GL font structure is almost identical to the Enhanced X-Windows font structure, each font registered with either the **loadXfont** or **defrasterfont** subroutine associates a handle with that font. GL keeps an internal hash table of all fonts registered for the process. Xfonts need to be registered with GL to be usable by GL.

Internal Properties

GL defines several Enhanced X-Windows properties that are used internally. If these properties are changed or destroyed, unpredictable behavior results. Therefore, use the following routines with caution:

- **XChangeProperty**
- **XDeleteProperty**

Widgets

The GL widget is not properly created and initialized if it is created with the **XtCreateManagedWidget** subroutine. To properly create and initialize the GL widget, use the **XtCreateWidget** subroutine and subsequently manage the widget with the **XtManageChild** subroutine.

Fullscreen Mode

In the current implementation of the GL **fullscrn** subroutine, the pointer is grabbed so that all input is redirected to the process that has called the **fullscrn** subroutine. Ungrabbing the pointer prematurely causes unpredictable results. Performing any other grabs or ungrabs can also cause unpredictable results within a **fullscrn**/**endfullscrn** subroutine pair.

The following Enhanced X-Windows functions interfere with GL fullscreen mode. Do *not* use these routines when GL is in fullscreen mode.

Enhanced X-Windows Grab Routines Restricted from GL Fullscreen Mode	
XChangeActive PointerGrab	**XUngrabServer**
XGrabButton	**XtGrabButton**
XGrabKey	**XtGrabKey**
XGrabKeyboard	**XtGrabKeyboard**
XGrabPointer	**XtGrabPointer**
XGrabServer	**XtRemoveGrab**
XUngrabButton	**XtUngrabButton**
XUngrabKey	**XtUngrabKey**
XUngrabKeyboard	**XtUngrabPointer**
XUngrabPointer	

Coordinate Transformation

GL and Enhanced X-Windows have different coordinate mappings. The GL window origin is in the lower left and the Enhanced X-Windows window origin is in the upper left. Do not use the **XTranslateCoordinates** subroutine, an Enhanced X-Windows function, to manipulate GL coordinate data.

Likewise, do not use GL matrix and viewport manipulation routines on Enhanced X-Windows windows.

Mixed GL and Windows Input

Currently, programming that combines GL input queue handling routines (such as **qread** and **qtest**) with X input queue handling routines (such as **XNextEvent** and **XCheckIfEvent**) on the same display connection is not recommended. X and GL input can be combined provided that such input is obtained through separate sockets.

Although you can use the Enhanced X-Windows event mechanism to obtain Enhanced X-Windows events for a GL window, do not try to use the GL event mechanism to obtain events for some GL windows and the Enhanced X-Windows event mechanism to obtain events for other GL windows.

The GL event queue is fed by events returning from the X server; the GL event handling library simply converts events from X format into GL format. The mapping is not one-to-one; sometimes multiple X events translate into only on GL event; sometimes multiple GL events are generated from one X event. If an unrecognized or unexpected event is encountered, it is discarded. Thus, changing the event mask for the GL connection, removing X events from the GL connection, or sending events to the GL connection results in the unpredictable or inconsistent operation of the GL input queue handling routines. It is for this reason that X and GL event routines should not be combined on the same display connection.

In the current implementation, when a **qtest** or **qread** subroutine call is made, all pending events for the process are retrieved from the X server and placed in an internal device queue. As a result, the GL and the Enhanced X-Windows event queues are no longer synchronized; that is, the arrival order of events is not preserved. Events retrieved by freely mixing calls to the GL **qread** subroutines and calls to the Enhanced X-Windows event processing routines are not guaranteed to be in chronological order.

In the current implementation, the GL **qenter** subroutine performs an **XSendEvent** subroutine call, and thus writes to the Enhanced X-Windows event queue.

The following is a partial list of Enhanced X-Windows functions that, if used in conjunction with the GL input queue, lead to asynchronous event delivery. Do *not* use these routines in conjunction with the GL input queue.

Enhanced X-Windows Event Routines Restricted from GL Input Queue	
XChangeKeyboardMapping	**XSelectInput**
XCheckIfEvent	**XSetPlaneMask**
XCheckMaskEvent	**XWindowEvent**
XCheckTypedEvent	**XtAppMainLoop**
XCheckTypedWindowEvent	**XtAppNextEvent**
XCheckWindowEvent	**XtAppPeekEvent**
XEventsQueued	**XtAppPending**
XIfEvent	**XtAppProcessEvent**
XNextEvent	**XtDispatchEvent**
XPeekEvent	**XtMainLoop**
XPeekIfEvent	**XtNextEvent**
XPending	**XtPeekEvent**
XPutBackEvent	**XtPending**
XQLength	**XtProcessEvent**

Example Programs

Example programs showing the usage of the **getXdpy**, **getXwid**, and **winX** subroutines can be found in the **/usr/lpp/GL/examples** directory.

Enhanced X-Windows and GL Interoperability

This section describes the interoperability between GL applications and Enhanced X-Windows and discussing interoperability issues such as queue handling and synchronization. The interoperability topics discussed in this section are as follows:

- Mapping and Unmapping GL Windows
- Integration of GL and Enhanced X-Windows
- Maintaining Synchronization
- X11 Header File Collision with the /usr/include/gl/gl.h File

Mapping and Unmapping GL Windows

The mapping of GL windows can be deferred by creating them using a **noport**; **winopen**; subroutine combination. The **mapwin.c** example program illustrates this usage. However, the following warning applies:

Attention: One should NEVER draw into a recently mapped window until after a REDRAW (MapNotify) event has been received for that window. For a window to be ready for GL rendering, it must not only be mapped, but the X server must complete a number of internal computations. After the X server has properly set up the window, it will generate a REDRAW (MapNotify) event for that window. Any drawing done prior to the receipt of the REDRAW event will result in a race condition: drawing orders may be lost or ignored, or other unrelated unpredictable behaviors may result (such as the inconsistent placement of the mapped window). Invoking XSync is NOT sufficient to guarantee that the X server has completed mapping the window.

Integration of GL and Enhanced X-Windows

By default, there is no guaranteed synchronization between actions performed through the Enhanced X-Windows programming interface (hereafter termed X) and the GL programming interface. In particular, there is no guarantee that drawing done with X and drawing done with GL will appear on the screen in the same order as performed by the application.

The underlying reason for this asynchronous behavior is the differing GL and X drawing models. The X server does all X drawing, and much of the rest of X processing. The processing does not actually occur until the X server is scheduled to run by the operating system; in the meantime, work requests are queued up. In contrast, all GL drawing subroutines send drawing orders to the graphics adapter directly; the adapter, in turn, renders as fast as possible.

Note: For some harware systems, there is no synchronization of drawing across GL windows resulting in asynchronous windows input.

Synchronization of X and GL drawing can be re-established by using the **finish** subroutine and the **XSync** subroutine. The **finish** subroutine will block (will not return) until all GL primitives have been rendered. The **XSync** subroutine flush the X display connection.

Use of the **XSync** subroutine does not guarantee that all X drawing is complete; it only guarantees that the Windows Server has received all generated Windows protocol packets. The Windows Server and/or adapter can still be in the process of rendering when the **XSync** subroutine returns. Since the latency between the Windows Server's receipt of a protocol packet and the completion of rendering to the screen is almost never more than a few milliseconds, this latency should normally not be noticeable.

Maintaining Synchronization

On the POWER Gt4 and the POWER Gt4x adapters, there is no guarantee that drawing is synchronized between different windows belonging to the same process. That is, a line drawn in one window may appear on the screen before a polygon is drawn in another window, even though the application made the subroutine call to draw the polygon before the subroutine call to draw the line. This de-synchronization occurs because a separate FIFO (first-in, first-out queue) is maintained for every window. All drawing commands are placed as tokens into the FIFOs. The graphics hardware has been designed to service the FIFOs in a round-robin fashion; the hardware does not necessarily drain one FIFO before moving on to the next. As a result, tokens in one FIFO may be processed before tokens in another, even though they were placed in the FIFO at a later time.

In most cases, the synchronization jitter should be on the order of milliseconds, or less, and should therefore be visually unobservable. If it is absolutely vital to re-establish synchronization, the **finish** subroutine can be used. The **finish** subroutine blocks (does not return to the calling application) until all buffers/FIFOs associated with the current window have been drained. The **finish** subroutine guarantees, in effect, that all drawing sent to a window has appeared on the screen.

X11 Header File Collision with the /user/include/gl/gl.h File

The **/user/include/gl/gl.h** file contains three typedefs that collide with X11 Toolkit Header files. These typedefs are as follows:

- typedef long Boolean;
- typedef long Object;
- typedef char *String;

To avoid this collision, adhere to following directions:
- Do not use these GL types in any source file that includes X11 code.
- Include all X11 header files before including any GL header file.

By doing this, the X11 types will hold, and the GL types will be undefined.

These three typedefs were in the **/usr/include/gai/g3dm2types.h** file and have been moved to the **/usr/include/gl/gl.h** file.

The typedefs Boolean and String were colliding with similiar typedefs in the **/usr/include/X11/Intrinsic.h** file while Object collided with a structure definition in the **/usr/include/X11/Object.h** file. The following was placed in **/usr/include/gl/gl.h** file to alleviate the type collisions.

```
#ifndef _XtIntrinsic_h

typedef long Boolean;

typedef char *String;

#endif

#ifndef _XtObject_h

typedef long Object;

#endif
```

This scenario requires that any include of XToolkit header files must occur before any include of **/usr/include/gl/gl.h** file. While using mixed mode programming the X11 types are the only valid types.

Chapter 14. Portability, Compatibility, and Performance

This section discusses the following topics:

- AIXwindows Environment/6000 3-D Feature Version 1 Release 3
- Example Programs
- Performance Tuning

AIXwindows Environment/6000 3-D Feature Version 1

In this release, GL is available to customers as a part of the optional 3-D feature to AIXwindows Environment/6000. This release of GL provides features that differ significantly from previous releases. These enhancements include new subroutines or changes to the operation of existing subroutines, new example programs, and new utilities, as follows:

charstr subroutine	Supports double-byte character set (DBCS) national language output.
dopup subroutine	Allows popup menus to operate in the overlay planes.
frontface subroutine	Enables frontfacing polygon culling on the POWER Gt4, POWER Gt4x, and the POWERgraphics GTO adapters.
gammaramp subroutine	Gamma ramps can be specified on a per window basis on the POWER Gt4, POWER Gt4x, and POWERgraphics GTO adapters. This function enables RGB mode partitions.
getgdesc subroutine	Returns graphics hardware description table.
getXdpysubroutine	Returns the Enhanced X-Windows connection of the GL session.
getXwidsubroutine	Returns the Enhanced X-Windows identifier of a GL window.
lmdef subroutine	Support spotlights. Support for two-sided lighting is available on the POWER Gt4 class adapters.
loadXfont subroutine	Allows users to access X fonts.
logicop subroutine	Supports additional arguments, which allow mathematical operations to be carried out on pixels, as well as logical operations. These arithmetic functions are supported on the POWER Gt4 and POWER Gt4x adapters.
makeobj subroutine	All rendering subroutines, including all begin and end style subroutines, can now be used inside an object.
pixmode subroutine	Pixmap row stride and pixel size can be specified on the POWER Gt4 and POWER Gt4x adapters.
winX subroutine	Converts Enhanced X-Windows into a GL window.
zwritemasksubroutine	Masks 8-bit banks of the z-buffer on the POWER Gt4 and POWER Gt4x adapters.
rendering subroutines	Most are now display-listable (can be used within objects). A complete list of these can be found in the GL Subroutine Modality table in Appendix B.
pop-up menus	Operate in the overlay planes. Refer to List of GL Pop-Up Menu Subroutines for descriptions of these subroutines.
chrate.c example program	Shows how the mouse transmit rate can be changed. This example program is located in the **/usr/lpp/GL/examples/chrate.c** file.

Note: When there is a conflict between GL standards and AIXwindows standards, the AIXwindows standard overrides the GL standard. If this affects the capabilities of a specific subroutine, a note is

included in the description section of that subroutine. (Subroutines are described in *GL3.2 for AIX: Graphics Library (GL) Technical Reference (POWER-based Systems Only)*.)

Example Programs

Additional example programs can be found in the **/usr/lpp/GL/examples** directory.

Note: Please consult the **README** file in the **/usr/lpp/GL** directory for additional information.

Performance Tuning

Programming in GL is similar to programming in a low-level language; small investments in programming effort can lead to large performance gains. Furthermore, different machines and adapters execute drawing commands at different relative drawing rates. An understanding of the type of hardware your code runs on can help you get optimal performance.

For example, there are many ways to draw a rectangle:

- Call the **scrmask** subroutine to mask off the appropriate area of the window, and then call the **clear** subroutine.
- Call the **czclear** subroutine.
- Call the **sboxf** subroutine.
- Load the appropriate transform matrices on the matrix stack, and then call the **rectf** subroutine.
- Call the **rectwrite** subroutine to block transfer of a pixmap that is one solid color.

Which of these methods draws the rectangle most quickly depends on the type of hardware on which the code is executing. The end goal also determines the choice of subroutines. If you want to clear the entire window and its z-buffer, the **czclear** subroutine is the fastest method of achieving this. But if you want to clear only a portion of the window, the **sboxf** subroutine executes faster than a combination of the **scrmask** and **clear** subroutines.

The previous discussion is only an example. There are several ways to perform nearly any graphics operation. Where speed is critical, so is design. The following paragraphs discuss some of the performance implications of new or modified subroutines.

lmcolor subroutine

If the design calls for changing material properties often, consider the **lmcolor** subroutine. By using the **lmcolor** subroutine, you can avoid using the relatively slowly executing combination of the **lmdef** and **lmbind** subroutines. If the **lmcolor** subroutine cannot be used, then perform all transactions with the **lmdef** subroutine before drawing. This method pulls the relatively slowly executing **lmdef** subroutine out of the performance-critical drawing loop.

curson, cursoff subroutines

In the current release of GL, the **curson** and **cursoff** subroutines turn the cursor visibility on and off. However, they execute fairly slowly and should not be heavily used.

GL formerly required the cursor to be turned off before any drawing was done, and turned on again after drawing completion. This is no longer required. However, existing code that is being ported to the current release may often make use of this function. This code should be modified to remove these subroutines, or the subroutines should be effectively removed by using the **#ifdef** condition with the C programming language preprocessor. Not doing so can result in a severe performance impact will be observed.

mapcolors, getmcolors subroutines

If the design employs color maps or color map mode, use the **mapcolors** and **getmcolors** subroutines. Using these subroutines improves performance 10 to 1,000 times over using the **mapcolor** or **getmcolor** subroutines inside a drawing loop.

Color maps are managed by the X server. The server updates the color map as change requests come into it. The **mapcolor** and **mapcolors** subroutines make one change request to the X server every time they are called. Because there is considerable system overhead in communication with the X server process, change requests are processed most efficiently if they are buffered and sent as one request by using the **mapcolors** subroutine.

concave subroutine

The **concave** subroutine causes the system to use more robust, but slower algorithms to render concave polygons correctly. If all affected polygons are convex, the applications programmer should disable the concavity check; this causes the system to use faster algorithms for polygon rendering. On most adapters, the performance improvement gained by disabling concave polygon-checking should be noticeable.

getcolor, getcpos, getgpos, getmatrix, gRGBcolor subroutines

These routines must query the graphics adapter to obtain the information that they return. Due to the current graphics adapter design, there is a significant amount of latency in such an operation. For the returned data to be valid, the graphics adapter must complete all preceding operations that may be queued for execution. In some cases, the queue may be quite deep, or some operations may take a long time. Therefore, the amount of time it takes any of these five routines to execute is variable, and could be quite long, depending on the contents of the queue. To obtain maximum performance, the use of these routines should be avoided.

scale subroutine

If lighting is enabled, and the **scale** subroutine is called with unequal arguments, the system is forced to renormalize all normal vectors. On some adapters, there is a noticeable performance improvement if the **scale** subroutine is not used with unequal arguments.

Writing Event Driven Applications

Most graphics applications are event driven. When developing such applications, keep in mind that the responsiveness is highly dependent on the rate of arrival of events. In particular, the mouse transmit rate can have a tremendous impact on *look and feel* performance (that is, how the system appears to respond to a mouse movement).

The reason for this is easy to understand. Many applications try to perform some fixed (and possibly large) amount of processing per mouse event. For example, the update and redraw of a slider, a scrollbar, or a 3D viewing parameter. High mouse transmit rates cause a large number of events to be queued up, and an application that does not discard some fraction of these events appears to be slow and sluggish. In contrast, low mouse transmit rates cause the same application to appear light and responsive.

Following are several remedies you can use to improve user-interface response.

- Decrease the mouse transmit rate. This is the most obvious approach. The program found in the **/usr/lpp/GL/examples/chrate.c** file shows how to use the terminal device driver to control the rate at which the mouse reports changes in its position.

  Note: Currently, there is no interface in GL, nor in AIXwindows, that controls this transmit rate.

  This remedy is acceptable for prototype and non-commercial code. However, if the application is meant to be marketed to a broad customer base, it is not wise to depend strongly on the mouse transmit rate. Setting the mouse transmit rate may adversely affect other applications; and conversely, if other applications set the mouse transmit rate, your application may be adversely affected.
- Decrease the amount of work done per event. One way to do this is to simply record the current mouse position whenever a MOUSEX or MOUSEY event arrives, but not perform any other work. When the event queue has finally been emptied, then the image (whatever is being drawn) can be updated based on the last recorded position. Thus, events are processed very quickly, but significant computation is not performed until events have stopped arriving.
- Discard every other, or even two out of every three, mouse events. This strategy is less desirable than the others because it can give an inferior look and feel. This strategy can be particularly troublesome when the user is attempting to position some object very carefully with the mouse. A small correction made by the user may be the event that is thrown away.

Minimizing REDRAW Events

You can reduce the number of REDRAW events issued to a GL application by changing the window manager default for feedback boxes. To do this, add the following to the **$HOME/.Xdefaults** file:

```
Mwm*showFeedback:restart
```

This setting causes the window manager not to display the small window size and position indicator that is normally shown when you move or resize a window. When it is on, this small indicator box (located in the center of the screen) can land on a GL window and generate REDRAW events. Multiple REDRAW events can result in significant performance impact on applications which take a long time to regenerate a window.

Fast Line Drawing

You can improve polyline drawing performance by using the **polylinelist** subroutine. The **polylinelist** subroutine helps eliminate the subroutine interface calling overhead of the `bgnline ... v ... v ... endline` interface. Subroutine calling overhead into the GL library can exceed one microsecond per call.

Fast Pixel Transfer (BLITS)

On the POWER Gt4 and POWER Gt4x adapters, and on the POWER Gt4, POWER Gt4x, and POWER GXT1000 adapters, the use of the **rectread** and **rectwrite**subroutines is deprecated. The **lrectread** and **lrectwrite** subroutines provide a faster, more efficient interface to pixel transfers. The format of the pixels can be adjusted with the pixmode subroutine.

Chapter 15. System Programming Considerations

This section discusses the following topics:

- Multiple Process Management
- Linking and Compiling Using the GL Shared Library
- Linking FORTRAN and C Modules
- Unsupported Subroutines
- Obsolete Subroutines

Multiple Process Management

Special considerations apply to the use of the **fork**, **vfork**, and other subroutines and to rendering processes used in conjunction with asynchronous handlers.

Using the fork, execl, execv and Other Subroutines

When using the **fork** or **vfork** subroutine with an Enhanced X-Windows, X Toolkit, or AIXwindows application, open a separate display connection (socket) with the **XOpenDisplay** or **XtOpenDisplay** subroutines for the forked process. The child process should never use the same display connection as the parent. Display connections are embodied with sockets and sockets are inherited by the child process. Any attempt to have multiple processes writing to the same display connection results in the random interleaving of X protocol packets at the word level. The resulting data written to the socket is generally invalid in the form of undefined X protocol packets that cannot be interpreted by the X server.

The **exec** and **fork** subroutines may be used within GL with some restrictions. The GL library manipulates a complex collection of resources. State information is stored in many places: in global (non-exported) variables in the process .data segment; in the X server; in the kernel, as part of the process table; and in the graphics hardware. Children created with the **fork** subroutine inherit (duplicate copies of) some, but not all of this state information. For example, the **fork** subroutine does not cause the X server to treat the child process as if it were a new client. In fact, the X server is ignorant of the fact that a fork has taken place. Similarly, the **exec** routine will overlay the old process image (in particular, wiping out values stored in global variables), but does not reset state contained in the X server or the kernel. (For example, the **exec** routine does not automatically close files or sockets.)

Current Restrictions

The following restrictions apply to the use of the **exec** and **fork** subroutines within a GL application:

- If the **exec** subroutine is used alone (that is, not immediately preceded by a call to the **fork** subroutine), and the new process image or any future overlays need to use GL, then the GL **gexit** subroutine must be called before the **exec** subroutine. This is required to reset process attributes; if this is not done, GL initialization performed by the new process image will be unsuccessful.
- If the **fork** subroutine is used alone (that is, not immediately followed by the **exec** subroutine) then the child process must not attempt to make any additional GL subroutine calls. Drawing by the child process may hang the graphics adapter or the X server, or may cause unpredictable results and place the system in an unpredictable state.

  In particular, the child process should not call the **gexit** subroutine. Because of a known implementation problem, the **gexit** subroutine adversely affects the parent process and the GL subroutine call in the child process.
- If the **fork** subroutine is used alone (that is, without a trailing **exec** subroutine) and both the child and the parent processes use GL, then the **gexit** subroutine must be called before the **fork** subroutine.
- If the **fork** subroutine is followed by an **exec** subroutine, the procedures for gracefully resetting the system depend on whether the **exec** subroutine occurs in the parent or in the child.

  If the **exec** subroutine is performed in the parent, then a **gexit** subroutine must precede the **exec** subroutine.

If the **exec** subroutine is performed by the child, then calling the **gexit** routine can be avoided, but at a risk. Failing to call the **gexit** subroutine leaves connections and shared memory segments open; this will prevent a process from performing more than nine consecutive forks in a row.

- For programs that end normally (for example, by using the **exit** subroutine), the **gexit** subroutine should be called first. In some circumstances, failure to do so can leave a process in the zombie state. Zombies are most likely to occur if the exiting process has child processes that are using GL subroutine calls.

Using Signals and Other Asynchronous Event Systems

GL is not designed to support rendering in a re-entrant fashion from within the same process. In particular, GL rendering from asynchronous handlers, such as signal handlers, is not recommended.

All GL rendering routines are interruptable; they do not mask any signals. GL rendering routines are particularly vulnerable when communicating with the graphics adapter. Adapter commands may be many words in length but must arrive in atomic units; adapter commands with embedded miscellaneous text or symbols may hang the adapter. If a GL routine is interrupted while writing to the adapter, and the interrupting routine also writes to the adapter, the resulting adapter command is embedded in the previously started data stream, thus corrupting it.

If the application performs rendering from within a signal handler, then the application must protect itself. This can be accomplished by masking all relevant signals before beginning rendering and unmasking them after completion. The application must mask all occurrences of rendering, whether within the signal handler or within the main program. Failure to do so risks corruption of the data stream and a hang of the graphic adapter.

Linking and Compiling Using the GL Shared Library

The GL library **libgl.a** is a shared library. By default, a GL program linked with the **-lgl** flag will link to the **libgl.a** library as a shared library. By using shared libraries, programs minimize the amount of disk space and memory that they use. A shared library is not bound in with the executable; thus the size of the executable is smaller (kilobytes, for small programs, instead of megabytes). Memory usage is decreased, because several programs running at the same time share the .text segment (the segment where the executable code is contained) of the library (a separate .data segment is created for each program).

> **Attention:** The GL library cannot be linked non-shared. Doing so will result in erratic and incorrect behaviour of GL programs. This warning applies to the C version of GL (**libgl.a** library) as well as the FORTRAN version (**libfgl.a** library).

Linking FORTRAN and C Modules

In general FORTRAN compilers append an underscore character ('_') to each user defined symbol. In this way name space conflicts can be avoided between C and FORTRAN. However, the xlf (f77) compiler does not add the trailing underscore to user defined symbols by default.

The FORTRAN GL library supports both naming conventions by exporting symbols with and without appended underscores. Because of a potential name space clash between the FORTRAN and C libraries, some care must be taken when linking together modules that use both the C and FORTRAN bindings for GL subroutines.

Programs written in FORTRAN and using GL fall into one of four categories. The following gives an example from each category, showing how FORTRAN GL applications can be linked.

To link FORTRAN-only source code, enter the following:

```
xlf -U foof.f -lfgl -lm
```

This binds the FORTRAN application with the FORTRAN GL library and the Math Library.

To link C and FORTRAN source where GL is referenced *only* through the FORTRAN interface, enter the following:

```
xlc -c fooc.c
xlf -U foof.f fooc.o -lfgl -lc -lm
```

This binds the MIXED application with the FORTRAN GL library, the C library, and the Math Library. The C Library is used to resolve externals from the module compiled by the **xlc** command.

To link C and FORTRAN source where GL is referenced only through the C interface, enter the following

```
xlf -U -c foof.f
xlc fooc.c foof.o -lgl -lxlf -lm
```

This binds the MIXED application with the C GL library, the XLF library, and the Math Library. The XLF Library is used to resolve externals from the module compiled by the **xlf** command.

To link C and FORTRAN source where GL is referenced through both the C and FORTRAN interfaces, enter the following:

```
xlf -U -qextname -c foof.f
xlc -c fooc.c
xlc  foof.o fooc.o -lgl -lc -lfgl -lxlf -lm
```

This creates the MIXED application. By compiling the FORTRAN modules with the **-qextname** flag, the extended naming rules are enforced. The extended naming rules append an underscore to all user-defined symbols.

1. The C libraries are specified first so that all C externals are resolved for the FORTRAN GL library contains references to functions with and without the trailing underscores. The FORTRAN libraries will then resolve all functions with the trailing underscores.
2. The order in which the libraries are placed on the compile line is important. If they are not placed according to the guidelines herein, the proper linkage will not be achieved and run-time errors will result.

Unsupported Subroutines

The following subroutines are not currently supported due to system constraints.

Note: These functions return without performing any function.

Unsupported Subroutines	
dglopen	**getvideo**
dglclose	**setvideo**
callfunc	**getmonitor**
e_callfunc	**setmonitor**
getresetls	**getothermonitor**
resetls	

When stippled lines are drawn (by calling the **setlinestyle** subroutine with a nontrivial style), the linestyle counter is normally initialized at the beginning of a polyline. As subsequent line segments are drawn, the linestyle counter is *not* reset, and the line stipple continues around the corner.

The **resetls** subroutine, now obsolete, set a flag (mode) that caused the system to reset the line stipple at every vertex. If it is important for your application to reset the linestyle for every line segment, surround each segment with a **bgnline** and **endline** subroutine pair. The **bgnline** subroutine automatically resets the line pattern counter when called.

On the POWER GXT1000 adapter, the line stipple is reset for every segment within a **bgn** and **end** subroutine pair.

Obsolete Subroutines

The following GL subroutines are no longer supported. The functions that they formerly accomplished are not applicable to the current system. As an aid to portability, stubs for these subroutines can be found in the **/usr/lpp/GL/examples/glports.h** file.

Obsolete Subroutines			
addtop	**getpor**	**pupmode**	**wintit**
charst	**getport**	**RGBcursor**	**xfpt**
clearhitcode	**gettp**	**RGBrange**	**xfpti**
dbtext	**gewrite**	**setdepth**	**xfpts**
devport	**gRGBcursor**	**setfastcom**	**xfpt2**
endfeedback	**iconti**	**setshade**	**xfpt2i**
endpupmode	**imakebackground**	**setslowcom**	**xfpt2s**
feedback	**ismex**	**shaderange**	**xfpt4**
foreground	**lsbackup**	**textcolor**	**xfpt4i**
getdepth	**pagecolor**	**textinit**	**xfpt4s**
gethitcode	**pagewritemask**	**textwritemask**	
getlsbackup	**passthrough**	**winattach**	

Chapter 16. Understanding the Graphics Adapter

GL graphics adapters have different capabilities and different sizes of frame buffers. The functions that control the frame buffer configuration behave differently depending on the size of the frame buffer and depending on which adapter is installed. The following discussion summarizes the available systems and adapters, the differences between them, and how these differences affect the behavior of GL.

GL is intended to be an interface to graphics hardware. If the graphics hardware (the graphics adapter) is not capable of performing certain functions, GL does not emulate these functions. Therefore, the available GL functions depend on the installed graphics adapter.

This section discusses the following topics:

- 3-D Color Graphics Processor
- RS/6000 POWERstation 730 and POWERgraphics GTO Supergraphics Processor Subsystem
- POWER Gt4 and POWER Gt4x Adapters
- POWER GXT1000 Adapter
- Hardware Colormap Organization

3-D Color Graphics Processor

The 3-D Color Graphics Processor is available in four configurations:

- 24-bit main color buffer with z-buffer
- 24-bit main color buffer without z-buffer
- 8-bit main color buffer with z-buffer
- 8-bit main color buffer without z-buffer

24-Bit High-Performance 3-D Color Graphics Processor with Z-Buffer Option

This adapter contains a 24-bit deep main frame buffer, 4 auxiliary planes dynamically configurable as overlay/underlay planes, and a 24-bit deep z-buffer. It contains a 4096-entry output color map (color lookup table) in onemap mode and sixteen 256-entry color maps in multimap mode.

In double buffer mode, the 24 main bitplanes are divided into a pair of buffers of 12 bits each. In RGB mode, these 12 bits are treated as 444 RGB, that is, as 4 bitplanes for storing red values, 4 bitplanes for storing green values, and 4 bitplanes for storing blue values. To maintain greater color fidelity and prevent color aliasing, dithering is automatically enabled, thus giving a greater perceived dynamic range (that is, more than 4 bits of accuracy for each band). The truncation of the nonleading bits and their convolution with the dithering matrix is performed automatically, and there is no modification of any of the drawing subroutines. The **lrectwrite** subroutine should be used for bit block transfers in RGB modes; it always assumes 888 RGB, and the appropriate bit-shifting to 444 is performed internally and automatically.

In double-buffered color map mode, all 12 bits of each buffer access the color map directly. Use the **rectread** and **rectwrite** subroutines for bit block transfers.

The 4 auxiliary bitplanes can be dynamically configured as 4 overlay planes, as 2 overlay and 2 underlay planes, or as 4 underlay planes.

Because this adapter does not have an alpha-blending buffer, the **blendfunction** subroutine returns without performing any action. Depth comparisons cannot be performed against the color buffer, therefore, the **zsource** subroutine returns without performing any action. The only valid value for the **zsource** subroutine is ZSRC_DEPTH.

24-Bit High-Performance 3-D Color Graphics Processor without Z-Buffer Option

This adapter contains a 24-bit deep main frame buffer and 4 auxiliary planes dynamically configurable as overlay/underlay planes. There is no z-buffer. It contains a 4096-entry output color map (color lookup table) in onemap mode and sixteen 256-entry color maps in multimap mode. The color maps are usable by the main frame buffer.

The main overlay/underlay and RGB/color map control functions operate identically as on the 24-bit processor with z-buffer option. The only difference between this adapter and the previously described adapter is that there is no z-buffer with this adapter. Therefore, none of the functions that control the z-buffer are operable; all return without performing any operation. These subroutines are as follows:

- **zbuffer**
- **zclear**
- **zdraw**
- **zfunction**
- **zsource**
- **zwritemask**

The action of the **czclear** subroutine is modified because this adapter does not have a z-buffer.

8-Bit High-Performance 3-D Color Graphics Processor with Z-Buffer Option

This adapter contains an 8-bit deep main frame buffer, 2 auxiliary planes dynamically configurable as overlay/underlay planes, and a 24-bit deep z-buffer. It contains a 256-entry output color map (color lookup table) in onemap mode and sixteen 16-entry color maps in multimap mode. The color maps are usable by the main frame buffer.

1. The 8-bit, high-performance 3-D Color Graphics Processor is limited in its ability to display multiple visible windows on the screen in conjunction with one or more GL double-buffered windows. It is possible for the user to establish enough visible windows on the screen so that it is impossible to display all the visible windows correctly and simultaneously. This limitation can manifest itself by displaying the wrong buffer, displaying a blank window, or displaying the window using the wrong colors. If the user needs to display multiple visible windows on the screen in conjunction with one or more GL double-buffered windows, the 24-bit High-Performance 3-D Color Graphics Processor should be used instead.
2. To avoid displaying windows improperly on the 8-bit adapter, the user should avoid running more than one GL window at the same time. If the user needs to run more than one GL window, the user should ensure that the window arrangement is rectangular, (for example, do not overlap windows in such a way that their boundaries cannot be described as a rectangle). Keeping windows rectangular also provides the user with faster window updates.

This adapter contains only 8 bits of main color buffer. It can be placed into RGB mode and into color map mode, and the operation of the subroutines that do this are unmodified.

In single-buffered RGB mode, pixels are written in 332 RGB: that is, with 3 bits of red, 3 bits of green, and 2 bits of blue. Dithering is always enabled. Because dithering uses the lesser significant bits to determine the color written into the frame buffer, the perceived accuracy of information stored in the frame buffer exceeds 8 bits and approaches 11 or 12 bits (as perceived by the human eye). (For more information on dithering, see Working in Color Map and RGB Modes.)

In single-buffered color map mode, all 8 bits are used to address the color map. In onemap mode, the color map is accessed as a single map of 256 entries. In multimap mode, it is accessed as 16 maps of 16 entries each.

This adapter supports double buffering only in color map mode. In this case, the main color buffer is divided into a pair of buffers of 4 bits each. In onemap mode, only the lowest 16 entries in the color map are addressed. When double buffered, this adapter should be operated in multimap mode. Double buffering in RGB mode is not available.

This adapter has 2 auxiliary planes, which can be configured as 2 overlay planes or 2 underlay planes.

This adapter does not have gamma ramps. The **gammaramp** subroutine is not emulated and returns without performing any function.

Because the sizes of the color maps supported on this adapter differ from those on the 24-bit adapter, the actions of the **onemap**, **multimap**, and **setmap** subroutines are modified.

Because this adapter does not have an alpha blending buffer, the **blendfunction** subroutine returns without performing any action. Depth comparisons cannot be performed against the color buffer; therefore, the **zsource** subroutine returns without performing any action. The only valid value for the **zsource** subroutine is ZSRC_DEPTH.

8-Bit High-Performance 3-D Color Graphics Processor without Z-buffer Option

This adapter contains an 8-bit deep main frame buffer, 2 auxiliary planes dynamically configurable as overlay/underlay planes. There is no z-buffer. It contains a 256-entry output color map (color lookup table) in onemap mode and sixteen 16-entry color maps in multimap mode. The color maps are usable by the main frame buffer.

The main, overlay/underlay, and RGB/color map control functions operate almost identically as those on the 8-bit with z-buffer option. The only difference between this adapter and the one previously described is that there is no z-buffer. Because there is no z-buffer, none of the functions that control the z-buffer are operable; all return without performing any action. The affected functions are as follows:

- **zbuffer**
- **zclear**
- **zdraw**
- **zfunction**
- **zsource**
- **zwritemask**

The action of the **czclear** subroutine is modified because this adapter does not have a z-buffer.

Because this adapter has no alpha blending buffer, the **blendfunction** subroutine returns without performing any action.

IBM RS/6000 POWERstation 730 and POWERgraphics GTO Supergraphics Processor Subsystem

The IBM RS/6000 POWERstation 730 has the Supergraphics Processor Subsystem adapter built in. This model has an extra-wide case to accommodate the Supergraphics Processor. The Supergraphics Processor Subsystem adapter cannot be removed, although other display adapters can be added. The POWERstation 730 is available in two standard configurations: with double-buffered 24-bit frame buffer and with double-buffered 8-bit frame buffer.

The IBM RS/6000 POWERgraphics GTO consists of a Supergraphics Processor Subsystem packaged in its own case with a power supply. A cable connects the POWERgraphics GTO to a Micro Channel adapter, which can be plugged into any IBM RS/6000. It is available in double-buffered 8-bit and double-buffered

24-bit configurations. The 24-bit POWERgraphics GTO is functionally equivalent to the IBM RS/6000 POWERstation 730 Supergraphics Processor Subsystem. The 8-bit POWERgraphics GTO differs from the POWERstation 730 in that it lacks the Shading Processor (ShP) and the z-buffer.

IBM RS/6000 POWERstation 730 and POWERgraphics GTO, 24-Bit Configuration

This configuration contains a total of 48 bitplanes in the main color frame buffer, 4 bits of overlay and a 21-bit z-buffer. There are multiple independent 256-entry color maps.

In single-buffered RGB mode, there are 24 bitplanes in the main color buffer, organized in an 888 RGB fashion. In double-buffered RGB mode, the front buffer and the back buffer each have 24 bitplanes. Dithering is not required and not supported.

In single-buffered color map mode, 8 bitplanes are available to select from a 256-entry color map. In double-buffered color map mode, there are 8 bitplanes in the front and back buffers each (a total of 16).

The RS/6000 POWERstation 730 and POWERgraphics GTO support a 21-bit z-buffer, and the operation of the z-buffer is modified. The following subroutines are affected:

lsetdepth
: The valid values for the parameters are 0x0 <= near <= far <= 0x1fffff.

greset
: The default depth range and shade range is from 0x0 to 0x1fffff.

zclear
: This function clears to the value that is farthest from the viewer for the default z comparison function. This value is 0x0.

zfunction
: The only valid values for the parameter are ZF_LEQUAL, ZF_GEQUAL, ZF_ALWAYS. The default depth comparison function is ZF_GEQUAL.

zsource
: The only valid value is ZSRC_DEPTH.

Note: The values written into the z-buffer are a nonlinear function of the z distance.

IBM RS/6000 POWERstation 730 and POWERgraphics GTO, 8-Bit Configuration

This configuration contains a total of 16 bitplanes in the main color frame buffer, 2 bits of overlay and a 21-bit z-buffer. There are multiple independent 256-entry color maps.

The 8-bit RS/6000 POWERgraphics GTO does not include the Shading Processor (ShP) and the z-buffer. Therefore, it does not support the Shading Processor, gouraud shading, lighting, depthcueing, blending, and z-buffering.

The 8-bit RS/6000 POWERstation 730 Supergraphics Processor Subsystem includes the ShP and the z-buffer, and therefore, does support the associated capabilities.

This configuration does not support RGB mode (and therefore does not support dithering). The only supported frame buffer modes are color map single buffered and color map double buffered.

In single-buffered color map mode, 8 bitplanes are available to select from a 256-entry color map. In double-buffered color map mode, there are 8 bitplanes each in the front and back buffers (a total of 16).

The POWERstation 730 and POWERstation support a 21-bit z-buffer, which operates as described in the 24-bit configuration.

Hardware Considerations

The RS/6000 Supergraphics Processor Subsystem contains specialized matrix multiplication hardware. As a result, the operation of the **loadmatrix** and **multmatrix** subroutines are modified. These subroutines accept only matrices of the following form:

Acceptable Matrices for Supergraphics Processor Subsystem			
a11	**a12**	**a13**	**k * a13**
a21	**a22**	**a23**	**k * a23**
a31	**a32**	**a33**	**k * a33**
a41	**a42**	**a43**	**a44**

Matrices of this type support all rotations and translations and most of the common perspective transformations.

The following constraints on patterns and linestyles apply to the Supergraphics Processor Subsystem:

Patterns Cannot be drawn and used if:

- Shademodel is set to Gouraud.
- Lighting is being used.
- The z-buffer is being used for depth comparisons.
- Depth-cueing is being used.

The only supported pattern sizes are 16x16 and 32x32. Affected subroutines are **defpattern**, **lmbind**, **setpattern**, **shademodel**, and **zbuffer**.

Lines Linestyle repeat factors must be a multiple of 4. Valid values run from 4 to 252 in multiples of 4. The lowest possible value for the linestyle repeat factor is 4. The affected subroutine is **lsrepeat**.

The following constraints apply to material properties on the Supergraphics Processor Subsystem:

Materials Material properties cannot be changed on a per-vertex basis. In particular, all calls to the **lmbind** and **lmcolor** subroutines made between **bgn...** and **end...** style primitives are ignored. This is true for triangular strips only and not polygons, points, and lines.

POWER Gt4 and POWER Gt4x Adapters

The POWER Gt4 and POWER Gt4x adapters are available in 16-bit (double-buffered 8-bit), and 48-bit (double-buffered 24-bit) configurations. Both adapters occupy 2 or 3 Micro Channel slots and support a 24-bit z-buffer.

Both adapters accelerate graphics line and polygon drawing speeds by using a custom very large scale integration (VLSI) raster subsystem, multiple digital signal processor (two on the POWER Gt4 and six on the POWER Gt4x), and a custom, high-speed VLSI interface chip. In addition to drawing speeds, these adapters optimize context switching (the servicing of drawing orders in multiple windows simultaneously) performance by supporting 16 independent logical first-in-first-out queues (FIFOs) and 16 graphics contexts resident on the adapter.

The POWER Gt4 and POWER Gt4x are functionally identical. The only difference between these two adapters is the number of onboard digital signal processors, which affects the overall drawing speeds. The POWER Gt4 and POWER Gt4x are available in two configurations:

- 48-bit (24 + 24) main color buffer, with 24-bit z-buffer
- 16-bit (8 + 8) main color buffer, with 24-bit z-buffer

The 48-bit POWER Gt4 and POWER Gt4x adapters each contain two 24-bit main frame buffers, 2 overlay bitplanes, and a 24-bit z-buffer. In double-buffered RGB mode, one 24-bit buffer is displayed, while the other is used for rendering. The POWER Gt4 and POWER Gt4x support multiple per-window hardware gamma ramps. In particular, while in RGB mode, the gamma ramp for one window can be changed independently of the others, thus leading to new visualization capabiltities. In color map mode, these adapters each support an 8-bit (256-entry) output color lookup table.

The 16-bit POWER Gt4 and POWER Gt4x adapters each contain two 8-bit main frame buffers, 2 overlay bitplanes, and a 24-bit z-buffer. When in double-buffer mode, one 8-bit buffer is displayed, while the other is used for rendering. In single-buffer mode, the single buffer is 8 bits deep. Double-buffered color-index and double-buffered RGB modes are both supported, the latter by 332 dithering. Like the 48-bit POWER Gt4 and POWER Gt4x, the 16-bit adapters support per-window gamma ramps. In color-map mode, these adapters each support an 8-bit (256-entry) output color lookup table.

The 24-bit POWER GXT1000 adapter is available in double-buffered 12-bit color-index or double-buffered 8-bit RGB mode; the 48-bit POWER GXT1000 adapter is available in double-buffered 12-bit color-index or double-buffered 24-bit RGB mode. Both adapters support 24-bit z-buffer and 8-bit overlay planes. Graphics rendering speeds are accelerated by both adapters by using a custom VLSI raster subsystem and multiple digital signal processors.

> Note: Valid z-buffer ranges on the POWER Gt4 and POWER Gt4x adapters are 0x0 to 0xffffff, and not the usual GL ranges -0x800000 to +0x7fffff.

Affected subroutines include the following:

- **getgdesc**
- **lrectread**
- **lrectwrite**
- **czclear**
- **lshaderange**
- **lRGBrange**
- **lsetdepth**
- **lgetdepth**

POWER GXT1000 Adapter

The POWER GXT1000 is an extended graphics adapter that emphasizes both high function and high mid-range performance. It provides hardware acceleration for 3D modeling and rendering, and addresses a diverse set of applications including design automation, architectural design, geophysical analysis, molecular modeling, physics, scientific visualizations, and animation. The POWER GXT1000 supports the following interfaces:

- AIXwindows
- OpenGL
- graPHIGS
- PEXlib
- GL

The POWER GXT1000 offers several visual sets (frame buffer configurations) that allow it to support a variety of graphical applications. There are two optional frame buffer features, the texture option and the advanced graphics option. With the base and optional features, the GXT1000 provides up to four visual sets:

- Base GXT1000
- Base GXT1000 + texture option

- Base GXT1000 + advanced graphics option
- Base GXT1000 + texture option + advanced graphics option

The POWER GXT1000 is a video stereo-ready graphics system that supports a method to add stereo 3D viewing (depth perception) to an image rendered on the 2D surface of a monitor. A stereo connector that provides the sync signal and + 12 V DC power is available on the back panel of the GXT1000. It allows attachment of an infrared transmitter that controls the image seen through liquid crystal lenses. This, along with a monitor that supports stereo mode, gives the effect of stereo 3D viewing. For more information, see the POWER GXT1000 Setup and User's Guide, Order Number SA23-2070-00.

> Note: To use GL on the GXT1000, the X server should be started with the `-x mbx` and `-x abx` flags. By default, the X server runs in the overlay planes. To start the server in the main frame buffer, use the `-layer 0` flag. The use of popup menus and the **fullscrn** subroutine can cause certain color flashing problems when the X server is run in the overlay planes.

Hardware Colormap Organization

Graphics hardware can be built to support several types of colormap organizations. There are two principal paradigms for colormap hardware: we refer to these as color index style and gamma ramp style. In AIXwindows X server terminology, these two styles are known as PseudoColor and DirectColor, respectively. Hardware can be built to support one or both of these styles, in a variety of combinations.

In color-index mode (PseudoColor mode), the frame buffer stores color index values which are used to look up RGB values in a color lookup table (CLUT). The color lookup table, or colormap, may be on a per-window or a per-screen basis; that is, it may affect the contents of only a particular window, or of the entire screen.

In RGB mode (DirectColor mode), the frame buffer stores RGB triplets. If the hardware supports gamma ramps, then each component of the triplet is passed through a separate lookup table. That is, the red value stored in the frame buffer is passed through the red lookup table; the result is a new value for the red component. Likewise, the green and blue components pass through their own separate tables. These types of color tables are often referred to as gamma-ramps, since they are often used to gamma-correct RGB color values. Note that PseudoColor mode can be emulated with DirectColor mode: if exactly the same value is written into the red, green, and blue channels of the frame buffer (and that value is the color index), and the color map is loaded into the red, green, and blue gamma ramps, then the resulting system is identical to a PseudoColor System. Note that gamma ramps may be built to operate on a per-window or a per-screen basis.

Graphics hardware may be built with either or both types of color table organizations. In the following sections, supported graphics hardware is described, and how GL subroutines can be used to set the color tables.

3-D Color Graphics Processor

The 24-bit High Performance 3D Color Graphics Processor contains per window color-index colormaps, and a single per-screen gamma ramp lookup table. In color index mode, the color-index table is used; in RGB mode, the color-index table is bypassed. In both modes, the gamma ramp tables are active and all RGB values, independent of their origin, pass through the gamma ramps. The **mapcolors** subroutine can be used to set the colormap; while the **gammaramp** subroutine can be used to set the gamma ramps.

On the 3D Color Graphics Processor, the cursor colormap is physically shared with the overlay color map. Changing the overlay colormap changes the color of the cursor. Changing the cursor colormap changes the overlay colormap. The X server manages the cursor, and that cursor colormap is reloaded whenever the cursor enters or leaves a window that has defined a special cursor.

EnterNotify and LeaveNotify events are generated by the X server whenever the cursor enters and leaves a window. Use the **XSelectInput** subroutine, with **EnterWindowMask**|**LeaveWindowMask** specified as the event mask.

POWERgraphics GTO

The 24-bit POWERgraphics GTO does not have any color-index colormaps. It does however, contain multiple gamma-ramp style look up tables, and these operate on a per-window basis. In color index mode, the **mapcolors** subroutine can be used to set these lookup tables. When in color-index mode, the operation of the POWERgraphics GTO is indistinguishable from the operation of an adapter with actual, physical color-index style colormaps. This is due to the power of a DirectColor architecture to emulate a PseudoColor architecture, as discussed above. In color-index mode, the **gammaramp** subroutine has no effect on the POWERgraphics GTO.

In RGB mode, the **gammaramp** subroutine can be used to set the lookup tables. The tables are set on a per-window basis; that is, setting the color table for one window does not affect the others. If the gamma ramp needs to be set for multiple windows, it must be done so for each window individually. Use the **winset** subroutine to set the current window, and then use **gammaramp** to set the color table. By default, whenever the **gconfig** subroutine is called to configure an RGB mode window, the lookup table is initialized to a linear ramp.

POWER Gt4 and POWER Gt4x

The operation of the 24-bit POWER Gt4 and POWER Gt4x subroutines with regard to colormaps and gamma ramps is identical to that of the 24-bit POWERgraphics GTO in every respect.

Chapter 17. GL Subroutines

The following list comprises all subroutines available in GL and the function of each.

Select an item from the following list:

GL Subroutines A - F

GL Subroutines G - L

GL Subroutines M - R

GL Subroutines S - Z

GL Subroutines (A-F)

You can find the subroutine you want by scrolling forward through this section, or by going directly to G-L, M-R, or S-Z.

Select an entry from the left column for reference information on the subroutine.

Select an entry from the right column for conceptual information relating to the subroutine.

addtopup	Adds an item to an existing pop-up menu.
arc	Draws a circular arc.
arcf	Draws a pie-shaped filled circular arc.
attachcursor	Attaches the cursor to two valuators.
backbuffer	Enables drawing in the back buffer.
backface	Allows or suppresses display of backfacing polygons.
bbox2	Culls and prunes to the bounding box.
bgnclosedline	Draws closed line vertices.
bgnline	Draws vertex-based lines.
bgnpoint	Draws vertex-based points.
bgnpolygon	Draws vertex-based polygons.
bgnsurface	Marks the beginning of a NURBS surface definition.
bgntmesh	Draws triangle mesh vertices.
bgntrim	Marks the beginning of a NURBS surface trimming loop.
blanktime	Sets screen blanking timeout.
blankscreen	Turns screen refresh on and off.
blendfunction	Specifies the alpha blending ratio.
blink	Changes the color map entry at a selectable rate.
blkqread	Reads multiple entries from the event queue.
c	Sets the current color in RGB mode.
callobj	Draws an instance of an object.
charstr	Draws a string of raster characters on the screen.
chunksize	Specifies the minimum object size in memory.
circ	Draws a circle.
circf	Draws a filled circle.
clear	Clears to the screenmask.
clkon/clkoff	Turn keyboard click on and off.
closeobj	Closes an object.
cmode	Sets color map mode as the current mode.
cmov	Moves the current character position.
color	Sets the current color in color map mode.

colorf	Sets the current color in color map mode (floating-point index).
compactify	Compacts memory storage of an object.
concave	Allows the system to draw concave polygons.
cpack	Sets the current color as a packed 32-bit integer.
crv	Draws a cubic spline curve.
crvn	Draws a series of cubic spline curves.
curorigin	Sets the origin of the cursor.
cursoff/curson	Control cursor visibility.
curstype	Defines the type and size of the cursor.
curvebasis	Sets the current cubic spline curve basis matrix.
curveit	Draws a curve segment by iterating the forward difference matrix.
curveprecision	Sets the number of line segments that compose a cubic spline curve.
cyclemap	Cycles between color maps at a specified rate.
czclear	Clears the color bitplanes and the z buffer simultaneously.
defbasis	Defines a cubic spline basis matrix.
defcursor	Defines the cursor.
deflinestyle	Defines a linestyle.
defpattern	Defines a pattern.
defpup	Defines a pop-up menu.
defrasterfont	Defines bitmaps for a raster font.
delobj	Deletes an object.
deltag	Deletes a tag from an object.
depthcue	Turns depth-cueing on and off.
dopup	Displays a pop-up menu.
doublebuffer	Sets the display mode to double buffer mode.
draw	Draws a line.
drawmode	Chooses a set of bitplanes for drawing.
editobj	Opens an object for editing.
endclosedline	Ends a series of closed line vertices.
endfullscrn	Ends full screen mode.
endline	Ends a series of vertex-based lines.
endpick	Turns off picking mode.
endpoint	Ends a series of vertex-based points.
endpolygon	Ends a vertex-based polygon.
endselect	Turns off selecting mode.
endsurface	Marks the end of a NURBS surface definition.
endtmesh	Ends a series of triangle mesh vertices.
endtrim	Marks the end of a NURBS surface trimming loop.
font	Selects a raster font.
freepup	Frees (deallocates) a pop-up menu and its data structures.
frontbuffer	Enables drawing in the front buffer.
frontface	Controls frontfacing polygon removal.
fudge	Specifies pixel values to be added to a window size.
fullscrn	Enables drawing outside the current window boundaries.

You can find the subroutine you want by scrolling back through this section, or by going directly to G-L, M-R, or S-Z.

Appendix A. GL Subroutines (G-L)

You can find the subroutine you want by scrolling forward through this section, or by going directly to A-F, M-R, or S-Z.

Select an entry from the left column for reference information on the subroutine.

Select an entry from the right column for conceptual information relating to the subroutine.

gammaramp	Defines a color map ramp for gamma correction.
gbegin	Initializes the graphics system without changing the color map.
gconfig	Reconfigures the system.
genobj	Returns a unique integer for use as object identifier.
gentag	Returns a unique integer for use as tag number.
getbackface	Indicates whether backfacing polygon removal is on or off.
getbuffer	Indicates which buffers are enabled for drawing.
getbutton	Returns the current state of a button.
getcmmode	Returns the organization of the current color map.
getcolor	Returns the current color in color map mode.
getcpos	Returns the current character position.
getcursor	Returns the cursor characteristics.
getdcm	Indicates whether depth-cue mode is on and off.
getdescender	Returns the baseline extent of the longest character descender.
getdev	Reads a list of valuators.
getdisplaymode	Returns the current display mode.
getdrawmode	Returns the current drawing mode.
getfont	Returns the current raster font number.
getfontencoding	Indicates the font encoding.
getfonttype	Indicates whether the current font is a double-byte character support (DBCS) font.
getgdesc	Returns information about currently installed graphics hardware.
getgpos	Gets the current graphics position.
getheight	Returns the maximum character height in the current raster font.
getlsrepeat	Returns the linestyle repeat count.
getlstyle	Returns the current linestyle.
getlwidth	Returns the current linewidth.
getmap	Returns the number of the current color map.
getmatrix	Gets a copy of the current transformation matrix.
getmcolor	Gets a copy of the RGB values for a color map entry.
getmcolors	Returns a range of color map RGB values.
getmmode	Returns the current matrix mode.
getnurbsproperty	Returns the current value of a trimmed NURBS surfaces display property.
getopenobj	Returns the current open object.
getorigin	Returns the position of a window.
getpattern	Returns the index of current fill pattern.
getplanes	Returns the number of available bitplanes.
getscrmask	Returns the current screenmask.
getsize	Returns the size of a window.
getsm	Returns the shading style used to draw filled polygons.
getvaluator	Returns the current state of a valuator.
getviewport	Gets a copy of the dimensions of the current viewport.
getXdpy	Returns the Enhanced X-Windows connection for the GL session.
getXwid	Returns the Enhanced X-Windows window ID for current GL window.
getwritemask	Returns the current writemask.
getzbuffer	Indicates whether z buffering is on or off.
gexit	Terminates a graphics program.
ginit	Initializes the graphics system.

greset	Resets all global state attributes to initial values.
gRGBcolor	Returns the current color (RGB mode).
gRGBmask	Returns the current RGB writemask.
gselect	Turns selecting mode on.
gsync	Waits for the next vertical retrace period.
gversion	Returns the version of GL being used.
iconsize	Specifies the size of a window icon.
icontitle	Specifies the title of a window icon.
initnames	Initializes the name stack.
isobj	Establishes the uniqueness of an object number.
isqueued	Indicates whether a specified device is enabled for event queuing.
istag	Establishes the uniqueness of a tag number.
keepaspect	Specifies the aspect ratio of a window.
lampon/lampoff	Turn the keyboard display lights on and off.
lgetdepth	Gets the distance of the near and far clipping planes.
linesmooth	Specifies antialiasing of lines.
linewidth	Specifies a linewidth.
lmbind	Binds a new material, light source, or lighting model definition.
lmcolor	Respecifies the currently bound material properties.
lmdef	Defines a new material, light, or lighting model.
loadmatrix	Loads a transformation matrix.
loadname	Loads a name on top of the name stack.
loadXfont	Loads an Enhanced X-Windows font into the font table.
logicop	Specifies a logical operation for pixel writes.
lookat	Defines a viewing transformation.
lrectread	Reads a rectangular array of pixels into host memory.
lrectwrite	Draws a rectangular array of pixels into the frame buffer.
lRGBrange	Sets the range of color indexes to use for depth-cueing in RGB mode.
lsetdepth	Sets the viewport depth range.
lshaderange	Sets the range of color indexes to use for depth-cueing in color map mode.
lsrepeat	Sets the repeat factor for the current linestyle.

You can find the subroutine you want by scrolling back through this section, or by going directly to A-F, M-R, or S-Z.

GL Subroutines (M-R)

You can find the subroutine you want by scrolling forward through this section, or by going directly to A-F, G-L, or S-Z.

Select an entry from the left column for reference information on the subroutine.

Select an entry from the right column for conceptual information relating to the subroutine.

makeobj	Creates a new object (display list).
maketag	Inserts a tag into the display list.
mapcolor	Changes a color map entry to a specified RGB value.
mapcolors	Loads a range of color map entries.
mapw	Maps a point on the screen into a line in 3-D world coordinates.

mapw2	Maps a point on the screen into a line in 2-D world coordinates.
maxsize	Specifies the maximum size of a window.
minsize	Specifies the minimum size of a window.
mmode	Sets the current matrix mode.
move	Moves the current graphics position to a specified point.
multimap	Organizing the color map as 16 small maps.
multmatrix	Premultiplies the current transformation matrix.
n3f	Specifies a normal vector for lighting calculations.
newpup	Allocates and initializes the structure for a new pop-up menu.
newtag	Inserts a tag at an offset from an existing tag.
noborder	Removes the border from a window.
noise	Filters valuator motion.
noport	Specifies that a program does not require a window.
normal	Specifies a normal vector for lighting calculations (can be used for display lists).
nurbscurve	Controls the shape of NURBS trimming curve.
nurbssurface	Controls the shape of an untrimmed NURBS surface.
objdelete	Deletes a routine from an object.
objinsert	Inserts a routine into an object.
objreplace	Replaces the existing display list routine with a new one.
onemap	Organizes the color map as one large map.
ortho	Defines a 3-D orthographic transformation.
ortho2	Defines a 2-D orthographic transformation.
overlay	Sets the number of bitplanes used for overlay.
patch	Draws a cubic spline surface patch.
patchbasis	Sets the current spline surface basis matrices.
patchcurves	Sets the number of curves used to represent a patch.
patchprecision	Sets the precision at which curves are drawn.
pclos	Closes a filled polygon.
pdr	Specifies the next point in a filled polygon.
perspective	Defines a perspective projection transformation in terms of a field of view.
pick	Puts the system in picking mode.
picksize	Sets the dimensions of the picking region.
pixmode	Controls operation of **lrectread** and **lrectwrite** subroutines.
pmv	Specifies the starting point for a filled polygon.
pnt	Draws a point.
pntsmooth	Specifies antialiasing of points.
polarview	Defines the viewer's position in polar coordinates.
polf	Draws a filled polygon.
poly	Draws a polygon.
polygonlist	Draws multiple, disjointed polygons.
polylinelist	Draws multiple, disjointed polylines.
popattributes	Pops the attribute stack.
popmatrix	Pops the transformation matrix stack.
popname	Pops a name off the name stack.
popviewport	Pops the viewport stack.
prefposition	Constrains the window position and size.
prefsize	Constrains the window size.
pushattributes	Pushes down the attribute stack.
pushmatrix	Pushes down the transformation matrix stack.

pushname	Pushes a new name onto the name stack.
pushviewport	Pushes the viewport onto the viewport stack.
pwlcurve	Describes a piecewise linear trimming curve for NURBS surfaces.
qdevice	Enables an input device for event queuing.
qenter	Creates an event queue entry.
qread	Reads the first entry in the event queue.
qreset	Empties the event queue.
qtest	Checks the contents of the event queue.
rcrv	Draws a rational cubic spline curve.
rcrvn	Draws a series of rational curve segments.
rdr	Draws a relative line.
readpixels	Returns a row of specific pixels in color map mode.
readRGB	Returns a row of specific pixels in RGB mode.
readsource	Specifies the source for pixels to be read.
rect	Draws a rectangle.
rectcopy	Copies a rectangle of pixels screen to screen with optional zoom.
rectf	Draws a filled rectangle.
rectread	Reads a rectangular array of pixels into host memory.
rectwrite	Draws a rectangular array of pixels into the frame buffer.
rectzoom	Specifies the zoom factor for rectangle copies and writes.
reshapeviewport	Sets the viewport to the dimensions of the current window.
RGBcolor	Sets the current color in RGB mode.
RGBmode	Sets a display mode that bypasses the color map.
RGBwritemask	Grants write permission to subset of available bitplanes (in RGB mode).
ringbell	Rings the keyboard bell.
rmv	Moves the current graphics position to a point relative to the current point.
rot	Rotates a graphical primitive (floating-point version).
rotate	Rotates a graphical primitive (fixed-point version).
rpatch	Draws a rational cubic spline surface patch.
rpdr	Draws a relative filled polygon.
rpmv	Moves the current graphics position to the starting point for a filled polygon relative the to current point.

You can find the subroutine you want by scrolling back through this section, or by going directly to A-F, G-L, or S-Z.

GL Subroutines (S-Z)

You can find the subroutine you want by scrolling forward through this section, or by going directly to A-F, G-L, or M-R.

Select an entry from the left column for reference information on the subroutine.

Select an entry from the right column for conceptual information relating to the subroutine.

sbox	Draws a screen-aligned rectangle.
sboxf	Draws a filled screen-aligned rectangle.
scale	Scales and mirrors objects.

screenspace	Interprets graphics positions as absolute screen coordinates.
scrmask	Defines a rectangular 2-D clipping mask.
setbell	Sets the duration of the keyboard bell sound.
setcursor	Sets cursor characteristics.
setdblights	Sets the lights on the dial and switch box.
setlinestyle	Selects a linestyle.
setmap	Selects one of 16 small color maps.
setnurbsproperty	Sets the property for display of trimmed NURBS surfaces.
setpattern	Selects a pattern for filling polygons and rectangles.
setpup	Enables or disables a given pop-up entry.
shademodel	Selects the shading style used to draw filled polygons.
singlebuffer	Sets the display mode to single buffer mode.
splf	Draws a shaded filled polygon.
stepunit	Specifies a window size change in discrete steps.
strwidth	Returns the width of the specified text string.
subpixel	Controls placement of point, line, and polygon vertices.
swapbuffers	Exchanges the front and back buffers.
swapinterval	Defines minimum time between buffer swaps.
swaptmesh	Toggles the triangle mesh register pointer.
swinopen	Creates a restricted subwindow.
textport	Allocates an area of the screen for a textport.
tie	Ties two valuators to a button.
tpoff	Turns off the textport.
tpon	Turns on the textport.
translate	Translates a graphical primitive.
underlay	Sets the number of bitplanes used for underlay.
unqdevice	Disables an input device for event queuing.
v	Transfers a vertex to the graphics pipe.
viewport	Set the area of the window used for all drawing.
winclose	Closes a window.
winconstraints	Binds window constraints to the current window.
windepth	Indicates the stacking order of windows on the screen.
window	Defines a perspective projection transformation in terms of *x* and *y* coordinates.
winget	Returns the identifier of the current window.
winmove	Moves the current window by its lower left corner.
winopen	Creates a new window.
winpop	Raises the current window on top of all other windows.
winposition	Changes current location and size of a window.
winpush	Lowers the current window beneath all other windows.
winset	Sets the current window.
wintitle	Adds a title bar to the current window.
winX	Converts an Enhanced X-Windows window into a GL window.
wmpack	Specifies an RGBA writemask with a single packed integer.
writemask	Grants write permission to a subset of available bitplanes in color map mode.

writepixels	Paints a row of pixels on screen in color map mode.
writeRGB	Paints a row of pixels on screen in RGB mode.
zbuffer	Enables or disables the z buffer for storing depth information.
zclear	Clears the z buffer.
zdraw	Enables drawing to the z buffer.
zfunction	Specifies the function used for depth comparison.
zsource	Selects depth or color as the source for z comparisons.
zwritemask	Specifies which bits of the z buffer are written during normal z buffer operation.

You can find the subroutine you want by scrolling back through this section, or by going directly to A-F, G-L, or M-R.

Chapter 18. GL Subroutine Modality

The following keywords from the subroutine modality list express applicable modalities for each GL subroutine.

Keyword	Indicates that the subroutine
display list	Can be used inside a display list (object).
current window	Affects only the current window.
push	Is affected by the **pushattributes** and **popattributes** subroutines.
bgn	Can be invoked between **bgn...**, **end...** subroutine calls.
winopen	Applies to next **winopen** or **winconstraints** subroutine call.
gconfig	Only takes effect when the **gconfig** subroutine is called.

Name	Effective Modality
addtopup	
arc	display list, current window
arcf	display list, current window
attachcursor	
backbuffer	display list, current window, push
backface	display list, current window
bbox2	display list (only)
bgnclosedline	display list, current window
bgnline	display list, current window
bgnpoint	display list, current window
bgnpolygon	display list, current window
bgnsurface	display list, current window
bgntmesh	display list, current window
bgntrim	display list, current window
blankscreen	
blanktime	
blendfunction	display list, current window
blink	
blkqread	
c	display list, current window, push, bgn
callobj	display list, current window
charstr	display list, current window
chunksize	
circ	display list, current window
circf	display list, current window
clear	display list, current window
clkon/clkoff	
closeobj	
cmode	current window, gconfig
cmov	display list, current window

Name	Effective Modality
color	display list, current window, push, bgn
colorf	display list, current window, push, bgn
compactify	
concave	display list, current window
cpack	display list, current window, push, bgn
crv	display list, current window
crvn	display list, current window
curorigin	
cursoff/curson	
curstype	
curvebasis	display list, current window
curveit	display list, current window
curveprecision	display list, current window
cyclemap	current window
czclear	display list, current window
defbasis	
defcursor	
deflinestyle	
defpattern	
defpup	
defrasterfont	
delobj	
deltag	
depthcue	display list, current window
dopup	
doublebuffer	current window, gconfig
draw	display list, current window
drawmode	current window, push
editobj	
endclosedline	display list, current window
endfullscrn	
endline	display list, current window
endpick	current window
endpoint	display list, current window
endpolygon	display list, current window
endselect	current window
endsurface	display list, current window
endtmesh	display list, current window
endtrim	display list, current window
font	display list, current window
freepup	

Name	Effective Modality
frontbuffer	display list, current window
frontface	display list, current window
fudge	winopen
fullscrn	
gammaramp	
gbegin	
gconfig	current window
genobj	
gentag	
getbackface	current window
getbuffer	current window
getbutton	
getcmmode	current window
getcolor	current window, push
getcpos	current window
getcursor	
getdcm	current window
getdescender	current window, push
getdev	
getdisplaymode	current window
getdrawmode	current window, push
getfont	current window, push
getfontencoding	current window, push
getfonttype	current window, push
getgdesc	
getgpos	current window
getheight	current window, push
getlsrepeat	current window, push
getlstyle	current window, push
getlwidth	current window, push
getmap	current window
getmatrix	current window
getmcolor	
getmcolors	
getmmode	current window
getnurbsproperty	current window
getopenobj	
getorigin	current window
getpattern	current window
getplanes	current window
getscrmask	current window

Name	Effective Modality
getsize	current window
getsm	current window, push
getvaluator	
getviewport	current window
getwritemask	current window, push
getXdpy	current window
getXwid	current window
getzbuffer	current window
gexit	
ginit	
greset	
gRGBcolor	current window, push
gRGBmask	current window, push
gselect	current window
gsync	current window
gversion	
iconsize	winopen
icontitle	winopen
initnames	display list, current window
isobj	
isqueued	
istag	
keepaspect	winopen
lampon/lampoff	
lgetdepth	current window
linesmooth	display list, current window
linewidth	display list, current window, push
lmbind	display list, current window, bgn
lmcolor	display list, current window, bgn
lmdef	
loadmatrix	display list, current window
loadname	display list, current window
loadXfont	
logicop	display list, current window
lookat	display list, current window
lrectread	current window
lrectwrite	current window
lRGBrange	display list, current window
lsetdepth	display list, current window
lshaderange	display list, current window
lsrepeat	display list, current window, push

Name	Effective Modality
makeobj	
maketag	
mapcolor	
mapcolors	
mapw	
mapw2	
maxsize	winopen
minsize	winopen
mmode	display list, current window
move	display list, current window
multimap	current window, gconfig
multmatrix	display list, current window
n3f	display list, current window, bgn
newpup	
newtag	
noborder	winopen
noise	
noport	winopen
normal	display list, current window, bgn
nurbscurve	display list, current window
nurbssurface	display list, current window
objdelete	
objinsert	
objreplace	
onemap	current window, gconfig
ortho	display list, current window
ortho2	display list, current window
overlay	current window, gconfig
patch	display list, current window
patchbasis	display list, current window
patchcurves	display list, current window
patchprecision	display list, current window
pclos	display list, current window
pdr	display list, current window
perspective	display list, current window
pick	current window
picksize	current window
pixmode	current window
pmv	display list, current window
pnt	display list, current window
pntsmooth	display list, current window

Name	Effective Modality
polarview	display list, current window
polf	display list, current window
poly	display list, current window
polygonlist	display list, current window
polylinelist	display list, current window
popattributes	display list, current window
popmatrix	display list, current window
popname	display list, current window
popviewport	display list, current window
prefposition	winopen
prefsize	winopen
pushattributes	display list, current window
pushmatrix	display list, current window
pushname	display list, current window
pushviewport	display list, current window
pwlcurve	display list, current window
qdevice	
qenter	
qread	
qreset	
qtest	
rcrv	display list, current window
rcrvn	display list, current window
rdr	display list, current window
readpixels	current window
readRGB	current window
readsource	current window
rect	display list, current window
rectcopy	current window
rectf	display list, current window
rectread	current window
rectwrite	current window
rectzoom	current window
reshapeviewport	current window
RGBcolor	display list, current window, push, bgn
RGBmode	current window, gconfig
RGBwritemask	display list, current window, push
ringbell	
rmv	display list, current window
rot	display list, current window
rotate	display list, current window

Name	Effective Modality
rpatch	display list, current window
rpdr	display list, current window
rpmv	display list, current window
sbox	display list, current window
sboxf	display list, current window
scale	display list, current window
screenspace	
scrmask	display list, current window
setbell	
setcursor	
setdblights	
setlinestyle	display list, current window, push
setmap	current window
setnurbsproperty	current window
setpattern	display list, current window, push
setpup	
setvaluator	
shademodel	display list, current window, push
singlebuffer	current window, gconfig
splf	display list, current window
stepunit	winopen
strwidth	current window, push
subpixel	display list, current window
swapbuffers	display list, current window
swapinterval	current window
swaptmesh	display list, current window
swinopen	
textport	
tie	
tpoff	
tpon	
translate	display list, current window
underlay	current window, gconfig
unqdevice	
v	display list, current window, bgn
viewport	display list, current window
winclose	
winconstraints	current window
windepth	
window	display list, current window
winget	

Name	Effective Modality
winmove	current window
winopen	
winpop	current window
winposition	current window
winpush	current window
winset	
wintitle	current window
winX	
wmpack	display list, current window, push
writemask	display list, current window, push
writepixels	current window
writeRGB	current window
zbuffer	display list, current window
zclear	display list, current window
zdraw	display list, current window
zfunction	display list, current window
zsource	display list, current window
zwritemask	display list, current window

Chapter 19. Adapter Description Table for GL

The following tables summarize the values returned by the **getgdesc** subroutine for each of the available adapters as follows:

Adapter Name	Column Name
8-bit 3-D Color Graphics Processor	8-bit 3-D
24-bit 3-D Color Graphics Processor	24-bit 3-D
16-bit Supergraphics Processor Subsystem	16-bit SPS
48-bit Supergraphics Processor Subsystem	48-bit SPS
16-bit POWER Gt4 and POWER Gt4x16-bit Gt4/Gt4x	
48-bit POWER Gt4 and POWER Gt4x48-bit Gt4/Gt4x	
24-bit POWER GXT1000	GXT1000
48-bit POWER GXT1000	GXT1000

The return is the value of the requested characteristic, or -1 if the request is not valid or cannot be determined. An empty entry in the table indicates that the characteristic is specific to the individual system configuration.

getgdesc Queries						
Inquiry	**8-bit 3-D**	**24-bit 3-D**	**16-bit SPS**	**48-bit SPS**	**16-bit Gt4/Gt4x**	**48-bit Gt4/Gt4x**
GD_ XMMAX	-1	-1	-1	-1	-1	-1
GD_ YMMAX	-1	-1	-1	-1	-1	-1
GD_ XPMAX	1280	1280	1280	1280	1280	1280
GD_ YPMAX	1024	1024	1024	1024	1024	1024
GD_ZMIN	-0x 800000	-0x 800000	0	0	-0x 800000	-0x 800000
GD_ZMAX	0x7fffff	0x7fffff	0x1fffff	0x1fffff	0x7fffff	0x7fffff
GD_BITS_ ALPHA- BUFFER	0	0	0	0	0	0
GD_BITS_ CURSOR	2	2	2	2	2	2
GD_BITS_ OVERLAY	2	4	2	4	0	2
GD_BITS_ PLANE_ MASK	0	0	0	0	0	0
GD_BITS_ UNDER LAY	2	4	0	0	0	0
GD_BITS_ ZBUFFER	24	24	21	21	24	24
GD_BITS_ NORM_ DBL_ CMODE	4	12	8	8	8	8
GD_BITS_ NORM_ DBL_ RGB	4	12	8	24	8	24
GD_BITS_ NORM_ SNG_ CMODE	8	12	8	8	8	8
GD_BITS_ NORM_ SNG_ RGB	8	24	8	24	8	24
GD_BIT_ OVER- UNDER_ SHARED	T	T	F	F	F	F
GD_ LARGE_ MAP_ SIZE	256	4096	256	256	256	256
GD_ SMALL_ MAP_ SIZE	16	256				

GD_NUM_ SMALL_ MAPS	16	16				
GD_ POINT- SMOOTH_ CMODE	T	T				
GD_ POINT- SMOOTH_ RGB	F	F				
GD_LINE- SMOOTH_ CMODE	T	T	F	F	T	T
GD_LINE- SMOOTH_ RGB	F	F	T	T	T	T
GD_ SHADE- MODEL						
GD_ SHADE- BUFFERS	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN
GD_ RGBMODE	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN
GD_ DITHER	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN
GD_SUBPIXEL_ LINE			0	0		
GD_SUB- PIXEL_ PNT			0	0		
GD_SUB- PIXEL_ POLY			0	0		
GD_POLY MODE	F	F	F	F	F	F
GD_BITS_ LINESTYLE	16	16	16	16	32	32
GD_MAX_ LSREPEAT	255	255	252	252	64	64
GD_MAX_ NURBS_ ORDER	4	4	8	8	8	8
GD_MAX_ TRIM_ ORDER	8	8	8	8	8	8
GD_MAX_ PATTERN _SIZE	64	64	32	32	64	64
GD_MAX_ VERTS	255	255	255	255	255	255
GD_MAX_ ATTR_ STACK- DEPTH	16	16	16	16	16	16
GD_MAX_ MATRIX_ STACK- DEPTH	32	32	64	64	32	32
GD_MAX_ VIEWPORT _STACK- DEPTH	8	8	8	8	8	8
GD_ ZDRAW_ GEOM	F	F	F	F	F	F
GD_ ZDRAW_ PIXELS	T	T	T	T	T	T
GD_ BLEND	F	F	T	T		
GD_ LIGHTING _TWOSIDE	F	F	F	F	T	T
GD_ BUTBOX						
GD_DIALS						
GD_VERT_RETRACE _FREQ	60	60	60	60	60	60
GD_ TEXTPORT	T	T	T	T	T	T
GD_ WSYS						

getgdesc Queries for 3-D and SPS Adapters				
Inquiry	**8-bit 3-D**	**24-bit 3-D**	**16-bit SPS**	**48-bit SPS**
GD_XMMAX	-1	-1	-1	-1

GD_YMMAX	-1	-1	-1	-1
GD_XPMAX	1280	1280	1280	1280
GD_YPMAX	1024	1024	1024	1024
GD_ZMIN	-0x800000	-0x800000	0	0
GD_ZMAX	0x7fffff	0x7fffff	0x1fffff	0x1fffff
GD_BITS_ ALPHA- BUFFER	0	0	0	0
GD_BITS_ CURSOR	2	2	2	2
GD_BITS_ OVERLAY	2	4	2	4
GD_BITS_ PLANE_ MASK	0	0	0	0
GD_BITS_ UNDERLAY	2	4	0	0
GD_BITS_ ZBUFFER	24	24	21	21
GD_BITS_ NORM_DBL_ CMODE	4	12	8	8
GD_BITS_ NORM_DBL_ RGB	4	12	8	24
GD_BITS_ NORM_SNG_ CMODE	8	12	8	8
GD_BITS_ NORM_SNG_ RGB	8	24	8	24
GD_BITS_ OVER- UNDER_ SHARED	T	T	F	F
GD_LARGE _MAP_SIZE	256	4096	256	256
GD_SMALL _MAP_SIZE	16	256		
GD_NUM_ SMALL_ MAPS	16	16		
GD_POINT- SMOOTH_ CMODE	T	T		
GD_POINT- SMOOTH_ RGB	F	F		
GD_LINE- SMOOTH_ CMODE	T	T	F	F
GD_LINE- SMOOTH_ RGB	F	F	T	T
GD_SHADE- MODEL				
GD_SHADE- BUFFERS	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN
GD_ RGBMODE	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN
GD_DITHER	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN	GD_BF_ MAIN
GD_ SUBPIXEL_ LINE			0	0
GD_ SUBPIXEL_ PNT			0	0
GD_ SUBPIXEL_ POLY			0	0
GD_ POLYMODE	F	F	F	F
GD_BITS_ LINESTYLE	16	16	16	16
GD_MAX_ LSREPEAT	255	255	252	252
GD_MAX_ NURBS_ ORDER	4	4	8	8
GD_MAX_ TRIM_ ORDER	8	8	8	8
GD_MAX_ PATTERN_ SIZE	64	64	32	32
GD_MAX_ VERTS	255	255	255	255
GD_MAX_ ATTR_ STACK- DEPTH	16	16	16	16
GD_MAX_ MATRIX_ STACK- DEPTH	32	32	64	64
GD_MAX_ VIEWPORT_ STACK- DEPTH	8	8	8	8

GD_ZDRAW_ GEOM	F	F	F	F
GD_ZDRAW_ PIXELS	T	T	T	T
GD_BLEND	F	F	T	T
GD_ LIGHTING_ TWOSIDE	F	F	F	F
GD_ BUTBOX				
GD_DIALS				
GD_VERT_ RETRACE_ FREQ	60	60	60	60
GD_ TEXTPORT	T	T	T	T
GD_WSYS				

getgdesc Queries for Gt4, Gt4x, and GXT1000 Adapters				
Inquiry	**16-bit Gt4/Gt4x**	**48-bit Gt4/Gt4x**	**24-bit GXT1000**	**48-bit GXT1000**
GD_XMMAX	-1	-1	-1	-1
GD_YMMAX	-1	-1	-1	-1
GD_XPMAX	1280	1280	1280	1280
GD_YPMAX	1024	1024	1024	1024
GD_ZMIN	-0x800000	-0x800000	0	0
GD_ZMAX	0x7fffff	0x7fffff	0xffffff	0xffffff
GD_BITS_ ALPHA-BUFFER	0	0	0	0
GD_BITS_ CURSOR	2	2	2	2
GD_BITS_ OVERLAY	0	2	8	8
GD_BITS_ PLANE_ MASK	0	0	0	0
GD_BITS_ UNDERLAY	0	0	0	0
GD_BITS_ ZBUFFER	24	24	24	24
GD_BITS_ NORM_DBL _CMODE	8	8	12	12
GD_BITS_ NORM_DBL _RGB	8	24	8	24
GD_BITS_ NORM_SNG _CMODE	8	8	12	12
GD_BITS_ NORM_SNG _RGB	8	24	8	24
GD_BITS_ OVER-UNDER_ SHARED	F	F	F	F
GD_LARGE _MAP_SIZE	256	256	4096	4096
GD_SMALL _MAP_SIZE				
GD_NUM_ SMALL_ MAPS				

GD_POINT-SMOOTH _CMODE			T	T
GD_POINT-SMOOTH _RGB			T	T
GD_LINE- SMOOTH_ CMODE	T	T	T	T
GD_LINE- SMOOTH_ RGB	T	T	T	T
GD_SHADE- MODEL				
GD_SHADE-BUFFERS	GD_BF_ MAIN	GD_BF_ MAIN		
GD_ RGBMODE	GD_BF_ MAIN	GD_BF_ MAIN		
GD_ DITHER	GD_BF_ MAIN	GD_BF_ MAIN		
GD_SUB-PIXEL_LINE				
GD_SUB-PIXEL_PNT				
GD_SUB- PIXEL_ POLY				
GD_POLY- MODE	F	F	F	F
GD_BITS_ LINESTYLE	32	32	16	16
GD_MAX_ LSREPEAT	64	64	255	255
GD_MAX_ NURBS_ ORDER	8	8	6	6
GD_MAX_ TRIM_ ORDER	8	8	6	6
GD_MAX_ PATTERN _SIZE	64	64	32	32
GD_MAX_ VERTS	255	255	?	?
GD_MAX_ ATTR_ STACK- DEPTH	16	16	16	16
GD_MAX_ MATRIX_ STACK- DEPTH	32	32	32	32
GD_MAX_ VIEWPORT _STACK-DEPTH	8	8	16	16
GD_ZDRAW _GEOM	F	F	F	F
GD_ZDRAW _PIXELS	T	T	T	T
GD_BLEND			F	F
GD_ LIGHTING _TWOSIDE	T	T	T	T
GD_ BUTBOX				
GD_DIALS				
GD_VERT_ RETRACE _FREQ	60	60	?	?

GD_ TEXTPORT	T	T	T	T
GD_WSYS				

Chapter 20. Porting SGI GL Applications to Your GL Environment

Graphics Library (GL) is a programming library interface that allows applications convenient access to the 3D graphics display hardware. Although your GL is developed to be compatible with the GL interface of Silicon Graphics, Inc. (SGI GL), it still requires some effort to port an application from the SGI GL environment to your GL environment.

Your implementation of GL is compatible with the SGI 4D Personal IRIS GL interface, but is not identical. This is due to the differences between hardware architecture, the operating system structure, and the development environment of the two platforms. For instance, GL is a command-oriented graphics system that allows the user's program to send any sequence of commands to the hardware. The behavior of the hardware is unspecified when invalid sequences are used. Because your system differs from SGI systems, they may behave differently in unspecified conditions.

The following procedures cover file transferring, compiling, and linking of topics as well as system performance and environment considerations.

- SGI GL File Transfer Compiling, and Linking
- SGI GL Performance and System Environment Considerations

SGI GL File Transfer Compiling and Linking

The Silicon Graphics workstations offered since1990 include: Personal IRIS, GT Graphics, POWERSeries 4D/200, and POWERSeries 4D/300. Data in each pair of bytes is restored in different order on these platforms than the order data is restored on your system. There are two ways to overcome this problem while transferring files:

1. Receive original archive files from the source platform, and swap bytes on the destination platform:
   ```
   dd if=/dev/rmt0 conv=swab | tar -xf -
   ```
2. Use the nonswap tape device when writing tapes on SGI systems. This will write a QIC 150 format tape on an IRIS machine, which then can be read by your system.
   ```
   tar -cf/dev/rmt0/tps0d7ns
   ```

To code a C program in your GL environment, the related GL include files should be specified in the program.

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

Application programs from a source platform usually invoke unsupported and obsolete subroutines. The **glport.h** file is provided for those who do not wish to clean up these old interfaces. Include the following statements in the application to declare these old subroutines void:

```
#define _GL_PORT_C_ 1
#include <gl/glport.h>
```

On SGI platforms, GL is shipped in two libraries: **libgl.a** and **libgl_s.a**. Linking to the first library results in a static link to GKL subroutines: the code for the GL routines is placed directly in the application binary, swelling its size considerably. The second library is a shared library. Linking to it does not pull GL library code into the application binary; rather, the shared library is dynamically loaded when the application is run. This results in a smaller application binary and in decreased memory usage when multiple GL applications are running simultaneously.

On your platform, there is only one library, **libgl.a**. This library is a shared library by default. One must not link a GL application nonshared on your equipment. On newer SGI GL releases, GL applications must link to both **libgl.a** and **libX11.a**, even though X11 is not being used directly. This is not required on your platform; such linkage is performed automatically.

The following GL subroutines are no longer supported. The functions that they formerly accomplished are not applicable to the current system. They are considered obsolete and should not be used in new code development. As an aid to portability, stubs for these subroutines can be found in the **/usr/lpp/gl/examples/gifts/glports.h** file.

Undefined Functions

dbtext **endpupmode**
pupmode **setfastcom**
setslowcom

Redefined Functions

```
#define getdepth lgetdepth
#define RGBrange IRGBrange
#define setdepth lsetdepth
```

Functions Returning Only One Value

endfeedback	0
gethitcode	0
ismex	TRUE

Functions Having Null Definitions

clearhitcode	devport
feedback	foreground
getlsbackup	getpor
getport	getresetls
gettp	gewrite
gRGBcursor	imakebackground
lsbackup	pagecolor
pagewritemask	passthrough
resetls	RGBcursor
setshade	shaderange
textcolor	textinit
textwritemask	winattach
xfpt	xfpti
xfpts	xfpt2
xfpt2i	xfpt2s
xfpt4s	xfpt4i
xfpt4	

Functions That Are Not Defined in libgl.a

dglopen	
dglclose	
callfunc	(define as null in **glport.h**)

e_callfunc
setvideo (define as null in **glport.h**)
getvideo (define as null in **glport.h**)
setmonitor (define as null in **glport.h**)
getmonitor (returns HZ60)
getothermonitor (returns HZ60)

SGI GL Performance and System Environment Considerations

Your display subsystem architecture is designed to support a wide range of application programming interfaces (API), such as graPHIGS, X11, and GL. Your GL subsystem is implemented in two pieces: GL windowing, input, and event queue functions are implemented as a module on top of **Xlib**; and GL drawing commands are sent directly to the graphics hardware. The internals of the 3D drawing subsystem are implemented in a high-performance, object-directed fashion, allowing GL applications to run transparently on different 3D graphics hardware without requiring recompilation or relinking. Additional information about your graphics subsystem architecture can be found in the Xhibition '91 Conference Proceedings (published by Integrated Computer Solutions, Inc. 201 Broadway, Cambridge MA 02139, xhibit@ics.com).

Performance characteristics of your GL implementation differs from the characteristics of the SGI GL implementation. Three system environments affect performance: the ″raw″ speed of the hardware, the software interface to the hardware, and the structure of application programs. Because some of your GL functions are supported by AIXwindows, these functions operate significantly slower in your GL, such as:

mapcolor **getmcolor**
viewport **scrnmask**
curson **cursoff**

To avoid using the above functions excessively, use the following substitutions to improve the performance of your GL implementation:

mapcolors **getmcolors**

Certain functions behave differently because of the operating system. The return of gversion is GL Version 3.2 on your platform while it is GL4DPI-3.2 or GL4DPI2-3.2 on the SGI platform.

The drawmode (PUPDRAW) call redirects the **mapcolor** and **getcolor** subroutines to affect the pop-up menus. In AIX 3.1, GL pop-up menus were implemented as separate X11 windows, on top of **Xlib**. This design caused the pop-ups to generate excessive REDRAW events to GL applications. To solve this problem, GL pop-up menus were redesigned in AIX 3.2 to operate in the overlay planes, using GL rendering calls.

In Color Map Mode, the value stored in the standard bit planes is interpreted as an index into a color map. With GL (AIXwindows environment), the order of the lowest 8 values was modified to be more compatible with X11: black was placed in slot 0 and white in slot 1. To be portable, these colors should be accessed with the #define'd token names found in <gl/gl.h> and never directly through their numeric value. The following table shows the color map's color values:

R	G	B	Color
0	0	0	Black
255	0	0	Red
0	255	0	Green

255	255	0	Yellow
0	0	255	Blue
255	0	255	Magenta
0	255	255	Cyan
255	255	255	White

With the SGI GL, the lowest 8 values in the default color map are loaded in a different order.

The font management facilities provided by SGI as part of the operating system are not part of your GL. The SGI FontManager library provides a font management system that allows scaling and rotating of annotation text fonts. The default font for the Personal IRIS has not been licensed by your supplier. Programs must choose fonts from those provided by the operating system and AIXwindows. To set default GL fonts for any programs inheriting the environment from a shell, execute the following:

```
GLFONT0=Rom10; export GLFONT0; glprog
```

All AIXwindows fonts are available for use in GL programs. Use the **XListFonts** call for an array of available font names, and then use **loadXfont** to load the specified font and return the font ID. Example programs demonstrating this usage can be found in **/usr/lpp/GL/examples**. With AIXwindows fonts available to your GL, most functions that are provided by SGI FontManager are satisfied.

In Version 3.2, **Xlib** calls can be mixed with GL calls in the same application. The guidelines for doing this, and the routines that enable this, can be found in Using Enhanced X-Windows Calls with GL Subroutines . Example programs demonstrating integration can be found in **/usr/lpp/GL/examples**.

Appendix A. Notices

This information was developed for products and services offered in the U.S.A.
IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.
IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.
This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.
The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and

cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Appendix B. Special Terms Used in GL

ambient light. Light that reflects off of one or more surfaces in the scene before arriving at the target surface. Ambient light is assumed to be non-directional, and is reflected uniformly in all directions by the reflecting surface. In the GL, ambient light is mocked up by use of ambient terms in the lighting equation, rather than actually computing the reflections.

aspect ratio. The ratio of the height of a primitive to its width. A rectangle of width ten inches and height five inches has an **aspect ratio** of 10/5 or 2.

asynchronous. Not synchronized in time. For example, input events occur at the choosing of the user—the program may read them later.

attribute. A parameter that can affect the appearance of a drawing primitive. For instance, color is an attribute. If the color is set to "RED", it will remain red until changed, and everything that is drawn will be drawn in red. Other attributes include linestyle, linewidth, pattern, and font. For a list of attributes and pipeline options, see the **greset** subroutine. See also *pipeline options***.**

azimuthal angle. If a primitive is sitting on the ground, with its *z* coordinate straight up, the azimuthal viewing angle is the angle the observer makes with the *y* axis in the *x-y* plane. If the observer walks in a circle with the primitive at the center, the azimuthal angle is the only thing that varies.

B-spline cubic curve. A cubic spline approximation to a set of four control points having the property that slope and curvature are continuous across sets of control points. See also *parametric cubic curve*.

backfacing polygon. A polygon whose vertices appear in clockwise order in screen space. If backface culling is enabled, such polygons are not drawn.

basis. In the GL, a curve or patch basis is a 4x4 matrix that controls the relationship between control points and the approximating spline. B-splines, Bezier curves, and Cardinal splines all differ in that they have different bases.

Bezier cubic curve. A cubic spline approximation to a set of four control points that passes through the first and fourth control points, and has a continuous slope where two spline segments meet. See also *parametric cubic curve*.

bitplane. A **bitplane** supplies one bit of color information per pixel on the display. Thus, an eight bitplane system allows 2 to the eighth power different colors to be displayed at each pixel.

blit. Bit block transfer.

Boolean. A value of TRUE or FALSE, where TRUE=1 AND FALSE=0

bounding box. A rectangle (2D) that bounds a primitive. A bounding box can be used to determine whether the primitive lies inside a clipping region. See *clipping.*

button. Buttons include those on the keyboard, mouse, light pen, or buttons on the dial and button box.

Cardinal cubic spline curve. A cubic spline whose endpoints are the second and third of four control points. A series of cardinal splines will have a continuous slope, and will pass through all but the first and last control points. See also *parametric cubic curve*.

clipping. If a primitive overlaps the boundaries of a window, it is *clipped*. The part of a primitive that appears in the window is displayed and the rest is ignored. There are several types of clipping that occur in the system. Three-D drawing primitives are clipped to the boundaries of a frustum (for perspective

transformations) or to a rhombohedron (for orthographic projections). This 3-D clipping applies as well to the origin of character strings, but not to the characters themselves. A 2-D clipping is also performed, where all drawing is clipped to the boundaries of AIXwindows. The area of 2-D clipping can be controlled with the screenmask. See *clipping planes*, *fine clipping*, *gross clipping*, *screenmask*, *transformation*, *window*.

clipping planes. Before clipping occurs, primitive space is mapped to normalized device coordinates. The clipping planes *x*= +-*w*; *y*= +-*w*; or *z*= +-*w* correspond to the left, right, top, bottom, near, and far planes bounding the viewing frustum. See frustum.

color map. A lookup table that translates color indexes into RGB triplets. The lookup table is sandwiched between the frame buffer and the digital-to-analog converters (DACs) and serves to translate the color index value stored in the frame buffer into the red, green, and blue values required by the DACs. On most hardware configurations, the color map is either 8 or 12 bits deep, allowing the simultaneous display of 256 or 4096 colors. On most hardware configurations, the DACs have an 8-bit per color accuracy, allowing the user to choose among 16,777,216 colors.

color map mode. A configuration of the hardware that passes the values stored in the frame buffer through a color lookup table (color map), from which the red, green, and blue values are obtained for display. Entries in the color map are referred to as color indexes. In color map mode, the values stored in the frame buffer are treated as color map indexes. See *RGB mode*.

color ramp. A progression of colors in a color map. Most color ramps are smooth, and have only a small number, if any, of discontinuities. For instance, if the full set of colors of the rainbow were loaded into the color map, that would constitute a color ramp.

concatenation. In the GL, concatenation refers to combining a series of geometric transformations—rotations, translations, and scaling. Concatenation of transformations corresponds to matrix multiplication.

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

control points. Points in real space that control the shape of a spline curve. The system provides hardware support for wire frame rational cubic splines, and for NURBS surfaces, the specifications of which require four control points.

culling. If a primitive is smaller than the minimum size specified in the command, it is *culled*: no further commands in the primitive are interpreted. See *clipping*, *pruning*.

current character position. The two-dimensional screen coordinates where the next character string or pixel read/write will occur.

current color. The color that is employed to color all subsequent drawing primitives. All drawing primitives are drawn with this color until it is changed.

current graphics position. The homogeneous three-dimensional point form which geometric drawing commands will draw. The current graphics position is not necessarily visible.

current transformation matrix. The transformation matrix on top of the matrix stack. All points passed through the graphics pipeline are multiplied by the current transformation matrix before being passed on. The current transformation matrix is a concatenation of the current modeling and viewing matrices. See *transformation*.

current window. The window to which the system directs the output from graphics routines. See also *window*.

cursor. A primitive such as an arrowhead which can be moved about the screen by means of an input device (typically a mouse).

cursor glyph. A 16x16 or 32x32 raster pattern (bitmap that determines the shape of the cursor. A GL cursor glyph can be one or two bits deep; thus, a GL cursor can use up to three colors. Color 0 is always transparent.

depth-cueing. Varying the intensity of a line with z-depth. Typically, the points on the line further from the eye are darker, so the line seems to fade into the distance.

device. A valuator, button, or the keyboard. Buttons have values 0 or 1 (up or down); valuators (mouse, dials) return values in a range, and the keyboard returns ASCII values.

dial and switch box. An I/O device with 8 dials (valuators) and 32 switches. The switch box is also called a *button box* or the *lighted programmable function keys* (LPFKs).

digital-to-analog converter (DAC). A highly specialized chip that converts the digital values coming out of the frame buffer into the rapidly varying analog voltage levels that are required by the monitor.

display list (object). Also called an object. It is a sequence of drawing commands that have been compiled into a unit. Conceptually, a display list is like a macro: it can be invoked multiple times simply by referring to its name. The object can be instantiated at different locations, sizes, and orientations by appropriate use of the transformation matrices. For instance, series of polygons arranged in the shape of a bolt can be compiled into an object. The bolt can then be drawn multiple times by invoking its display list.

dithering. A technique of interleaving dark and light pixels so that the resulting image looks smoothly shaded when viewed from a distance.

double buffer mode. A mode in which two buffers are alternately displayed and updated. A new image can be drawn into the back buffer while the front buffer (containing the previous image) is displayed. See *single buffer mode*.

event queue. A queue that records changes in input devices—buttons, valuators, and the keyboard. The event queue provides a time-ordered list of input events.

eye coordinates, eye space. The coordinate system in which the viewer's eye is located at the origin, and thus all distances are measured with respect to the eye. Viewing transformations map from world coordinates into eye coordinates, and projection transformations map from eye coordinates to normalized device coordinates. Also called viewing coordinates or viewer coordinates. See *modeling coordinates*, *world coordinates*, *screen coordinates*, *transformation*.

field of view. The extent of the area which is under view. The field of view is defined by the **viewing matrix** in use.

fine clipping. Fine clipping masks all drawing commands to a rectangular region of the screen. It would be unnecessary except for the case of character strings. The origin of a character string after transformation may be clipped out by gross, or 3-D, clipping, and the string would not be drawn. By doing gross clipping with the viewport and fine clipping with the screen masks, strings can be moved smoothly off the screen to the left or bottom. See *gross clipping*.

font. A set of characters in a particular style. See *raster font*, *primitive font*.

forward difference matrix. A 4x4 matrix that is iterated by adding each row to the next and the bottom row is output as the next point. Points so generated generally fall on a rational cubic curve.

frame buffer. A quantity of video RAM (VRAM) that is used to store the image displayed on the monitor.

The frame buffer is the electronic canvas on which every drawing primitive is drawn. It is one of the last stops in the graphics pipeline, where the final image resides in the form of digitally coded intensities and brightnesses. These are converted into analog voltage signals 60 times a second and sent to the electron guns of the monitor.

The dimensions of the frame buffer can be changed with GL. Typically, the main frame buffer might be 1024 pixels vertical by 1280 pixels horizontal by 8 color bits. The overlay planes might by 1024x1280x2. The z-buffer is considered a frame buffer, although it is not directly visible from the monitor. (There is no direct means of displaying the contents of the z-buffer, although this can be done indirectly.) The size of the z-buffer is typically 1024x1280x24. The cursor is a very specialized form of a frame buffer; one which can move around. The typical cursor is 32x32x2 in size.

front and back buffers. In double buffer mode, the main frame buffer bitplanes are separated into two sets—the front and back buffers. Bits in the front buffer planes are visible and those in the back buffer are not. Typically, an application draws into the back buffer and views the front buffer for dynamic graphics.

frustum. A truncated, four-sided pyramid; that is, a pyramid with the point cut off. In a perspective projection, the shape of the clipping volume is a frustum. The bottom of the frustum is referred to the far clipping plane, the top of the frustum is the near clipping plane, and the sides are respectively the top, left, bottom, and right clipping planes. In an orthographic projection, the clipping volume is a parallelepiped.

gamma correction. A logarithmic assignment of intensities to lookup table entries for shading applications. This is required since the human eye perceives intensities logarithmically rather than linearly.

gamma ramp. A set of three lookup tables, one for each of the colors red, green, and blue, attached to the electron guns of the monitor. Entries in the gamma lookup table can be set to adjust for variations in the phosphor quality between different brands of monitors. Usually, a logarithmic curve is loaded into the gamma lookup tables. (See *gamma correction*.) The gamma lookup tables are not a subset of the color map tables, but a separate entity.

Gouraud shading. A method of shading polygons smoothly based on the intensities at their vertices. The color is uniformly interpolated along each edge, and then the edge values are uniformly interpolated along each scan line. For realistic shading, colors should be gamma corrected.

graphics pipeline. The sequence of steps that a graphics primitive goes through before it becomes visible on the screen:

- Transformation from model coordinates to NDC coordinates
- 3-D clipping (if out of bounds)
- Perspective division
- Determination of color through lighting equations or depth-cueing
- Transformation of NDC coordinates to screen coordinates
- 2-D clipping (by the screenmask)
- Rasterization (drawing into the frame buffer)
- Display of frame buffer.

gross clipping. Also known as 3-D clipping this is the clipping that occurs in normalized device coordinates, against the sides of the perspective frustum. All 3-D primitives undergo this clipping; in particular, the origin of text strings (but not individual letters) are clipped in this way. See *clipping planes*, *fine clipping*.

hidden surface. A surface of a geometric primitive that is not visible because it is obscured by other surfaces. See *z-buffering*.

hit. Also called **pick hit** or **select hit.** A hit occurs whenever a drawing primitive draws within the picking or selecting region. A hit is reported back to the user only if the name stack has changed since the last hit. In other words, multiple hits may occur although only one pick/select even is reported. See *name stack*, *picking*, *selecting*.

homogeneous coordinates. A four-dimensional method of representing three-dimensional space. A point (*x*, *y*, *z*, *w*) in homogeneous coordinates is used to represent a point (*X*, *Y*, *Z*) in three-dimensional space by taking *X*=*x*/*w*, *Y*=*y*/*w*, *Z*=*z*/*w*.

immediate mode. In this mode, graphics commands are executed immediately rather than being compiled into a display list.

instantiate. To make an instance of. To replicate.

linear interpolation. A method of approximating data values by assuming that they lie along a straight line. Typically, the two end data points are known. For example, if A is the value at a, and B is the value at b, and a<t<b, then the value C at t is (from the two-point formula):

$$C(t) = \frac{(B - A)}{(b - a)} (t - d) + A$$

linestyle. The pattern used to draw a line. A linestyle might be solid or broken into a pattern of dashes.

linewidth. The width of a line in pixels.

matrix stack. A stack of matrices with hardware and software support. The top matrix on the stack is the current transformation matrix, and all points passed through the graphics pipeline are multiplied by that matrix. It is a concatenation of the current modeling and viewing transformations. See *current transformation matrix*.

mirroring. The creation of a mirror image of a primitive.

modeling coordinates, modeling space. The coordinate system in which all drawing primitives do their drawing. The user can select the position and orientation of the modeling space with regard to the world space by means of translations, rotations, scales, or generalized transformations. The relation between modeling coordinates and world coordinates is determined by the modeling matrix. Modeling coordinates are a useful conceptual device when drawing complex or repetitive scenes. For instance, a paper clip can be defined once in modeling coordinates, and then drawn hundreds of times by moving the modeling coordinates around in world space. See *eye coordinates*, *screen coordinates*, *world coordinates*, *transformation*.

name stack. A stack of 16-bit integers, controllable by the user, used to establish what drawing primitive caused a pick or select event. The name stack is written into the pick/select event buffer every time a pick or select event occurs. The entire event buffer is returned to the user at the end of the pick/select episode.

normalized device coordinates (NDC). Coordinates in the range from -1 to 1. All primitives that draw within the unit cube are visible on the screen (unless masked by the screenmask). See *transformation*, *unit cube*.

NTSC. A video display and timing format that is the American broadcast standard. Most video tape recorders record and play back NTSC signals. Specialized hardware is required to convert from RGB monitor outputs to an NTSC signal.

null-terminated. Having a zero byte at the end. In the C language, character strings are stored this way internally.

NURBS. (Non-Uniform Rational B-spline). A parametric surface that can be trimmed with non-uniform rational B-spline curves and piecewise linear curves.

object. Also called a display list. It is a sequence of drawing commands that have been compiled into a unit. Conceptually, a display list is like a macro: it can be invoked multiple times simply by referring to its name. The object can be instantiated at different locations, sizes, and orientations by appropriate use of the transformation matrices. For instance, series of polygons arranged in the shape of a bolt can be compiled into an object. The bolt can then be drawn multiple times by invoking its display list.

object space. The space in which a graphics object is defined. A convenient point is chosen as the origin and the object is defined relative to this point. When an object is rendered by a call to the **callobj** subroutine, it is rendered in modeling coordinates, and the object space becomes (for that moment) the same as the modeling space.

orthographic projection. A representation in which the lines of a projection are parallel. Orthographic projections lack *perspective foreshortening* and its accompanying sense of depth realism. Because they are simple to draw, orthographic projections are often used by draftsmen. See *perspective projection*.

parametric bicubic surface. A surface defined by three equations. The xequation is:

$$x(u,v) = a_{11}u^3v^3 + a_{12}u^3v^3 + a_{13}u^3v + a_{14}u^3$$

$$+ a_{21}u^2v^3 + a_{22}u^2v^2 + a_{23}u^2v + a_{24}u^2$$

$$+ a_{31}uv^3 + a_{32}uv^2 + a_{33}uv + a_{34}u$$

$$+ a_{41}v^3 + a_{42}v^2 + a_{43}v + a_{44}$$

The equations for *y* and *z* are similar.

The points on a bicubic patch are defined by varying the parameters *u* and *v* from 0 to 1. If one parameter is held constant and the other is varied from 0 to 1, the result is a cubic curve. If $w(u,v)=1$ for all u,v, the bicubic surface is called “ordinary,” but if $w(u,v)$ varies as a function of u,v, then the surface is called “rational.” See also *homogeneous coordinates*.

parametric cubic curve. A curve defined by the equation:

$$x(t) = a_xt^3 + b_xt^2 + c_xt + d_x$$

$$y(t) = a_yt^3 + b_yt^2 + c_yt + d_y$$

$$z(t) = a_zt^3 + b_zt^2 + c_zt + d_z$$

$$w(t) = a_wt^3 + b_wt^2 + c_wt + d_w$$

where *x, y, z,* and *w* are cubic polynomials. The parameter *t* typically varies between 0 and 1. Such a curve is considered rational only if a(w), b(w), or c(w) is not equal to 0; otherwise, it is simply an ordinary parametric curve.

patch. A parametric bicubic surface.

pattern. A 16x16, 32x32, or 64x64 array of bits defining the texturing of polygons on the system display.

perspective projection. Perspective projection is a technique used to achieve realism when drawing primitives. In a perspective projection, the lines of projection meet at the viewpoint; thus the size of a primitive varies inversely with its distance from the source projection. The farther a primitive or part of a primitive is from the viewer, the smaller it will be drawn. This effect, known as *perspective foreshortening*, is similar to the effect achieved by photography and by the human visual system. See *orthographic projection*.

picking. A method for finding out what primitives are being drawn near the cursor on the display screen. See *picking region*, *selecting*, *selecting region*, *name stack*.

picking region. A rectangular volume around the cursor that is sensitive to picking events. If a drawing primitive draws within this volume, a pick event is reported. The width and height of the region can be set by the user. If the z-buffer is enabled, the depth of the region is the entire z-buffer. See *hit*, *picking*, *selecting region*.

piecewise linear curve. A list of coordinate pairs in the parameter space for the non-uniform rational B-spline (NURBS) surface. These points are connected with straight lines to form a path.

pipeline options. Variables that control the flow of processing in the graphics pipeline. For instance, lighting is a pipeline option: if lighting is turned on, the color of a primitive is obtained by evaluating the lighting equations, and if lighting is turned off, the last color specified is used. Other pipeline options are the backfacing flag, the shademodel flag, the depth-cueing flag, the picking flag, the color mode (color index or RGB) flag, the z-buffer flag (enables or disables drawing to the z-buffer), and so on. See also *attributes*.

pixel. A rectangular picture element. A display screen is composed of an array of pixels. In a black-and-white system, pixels are turned on and off to form images. In a color system, each pixel has three components: red, green, and blue. The intensity of each component can be controlled.

pixmap. A rectangular array of pixels. The rectangle may be of any size. The format of a pixel is arbitrary and may be an RGB triplet or a color index.

polar coordinates. A coordinate system in which positions are measured as a distance from the origin and an angle from some reference direction (usually, counterclockwise from the *x*-axis).

polled I/O devices. Devices (keyboard, mouse, button, dials) whose current values are read by the user process.

pre-multiplication. Matrix multiplication on the left. If a matrix M is pre-multiplied by a matrix T, the result is TM.

precision. The number of straight line segments used to approximate one segment of a spline.

primitive. A drawing command, such as **arc**, **line**, **circle**, **polygon**, or **charstr**. Such commands are called primitives because they are not made up of smaller parts, and because they are they basic pieces out of which more complex scenes can be composed. Also used to describe the figures created by drawing commands.

primitive font. A font in which characters are defined as primitives. Like all other primitives, primitive font characters can be scaled and rotated. See *raster font*.

pruning. Eliminating the drawing of parts of the display list because a bounding box test shows that they are not visible.

queued I/O devices. Devices (keyboard, mouse, button, dials) whose changes are recorded in the event queue.

raster font. A font in which the characters are defined directly by the raster bit map. See *font*, *primitive font*.

raster subsystem. That part of the subsystem concerned with an image after it has been transformed and scaled to screen coordinates. It includes scan conversion and display.

refresh rate. The rate at which the monitor is refreshed. A 60 Hz monitor is redrawn 60 times per second.

relative drawing commands. Commands that draw relative to the current graphics position as opposed to being drawn at absolute locations.

repeat factor. The magnification with which the linestyle pattern is used.

RGB mode. A configuration of the hardware which allows values stored in the frame buffer to be interpreted as packed RGB values. The values found in the frame buffer are passed directly to the red, green, and blue guns of the display monitor. The values are not passed through the color map first. (However, each color is sent individually through the gamma ramp to make a final correction to its intensity.) See *color map mode*.

RGB value. The set of red, green, and blue intensities that compose a color is that color's RGB value.

RGBA value. The set of red, green, blue, and alpha intensities that compose a color is that color's RGBA value. Alpha values are available only on machines having alpha bitplanes.

right-hand rule. If the right hand is wrapped around the axis of rotation, the fingers curl in the same direction as positive rotation, and the thumb points in the same direction as the axis of rotation. A right-handed rotation is counter-clockwise.

rotation. The transformation of a primitive by rotating it about an axis. See *transformation*.

scaling. Uniform stretching of a primitive along an axis.

screen coordinates, screen space. The coordinate system that defines the display screen. Distances are measured in units of pixels, and the origin is in the lower left-hand corner. On most systems the screen size is 1024 pixels high by 1280 pixels wide. The viewport defines the mapping from normalized device coordinates to screen coordinates. See *eye coordinates*, *modeling coordinates*, *world coordinates*, *transformation*.

screenmask. A rectangular area of the screen to which all drawing operations are clipped. It is normally set equal to the viewport and to the window. A screenmask is useful for character clipping.

selecting. A method for finding what primitives are being drawn in a given volume in 3-D space. See *picking*, *picking region*, *name stack*, *selecting region*.

selecting region. A rhomboid-shaped volume in world coordinates that is sensitive to selecting events. If a drawing primitive draws within this region, a select event is reported. See *hit*, *picking region*, *selecting*, *transformation*.

single buffer mode. A mode in which the frame buffer bitplanes are organized into a single large frame buffer. This frame buffer is the one currently displayed and is also the one in which all drawing occurs. See *double buffer mode*.

swap interval. The amount of elapsed time between frame buffer swaps. The system waits at least the amount of time specified by the **swapinterval** subroutine before honoring a request to exchange the front and back buffers. The swap interval is measured in units of vertical retraces, which occur every 30th of a second on most systems. The swap interval is useful in achieving smooth-flowing animation.

tag. A marker in the display list used as the location for display list editing.

textport. A region on the display screen used to present textual output from graphical or non-graphical programs.

texture. A pattern used to fill rectangles, convex polygons, arcs, and circles.

transformation. A four-by-four matrix that helps determine the location where 3D drawing will occur, the position of the viewpoint (the viewer's eye), and the amount of the scene encompassed and visible. Transformations occur at four points within the graphics pipeline:

- Modeling transformation, which maps modeling coordinates into world coordinates. All drawing primitives specify positions that are presumed to be positions in modeling coordinates. Modeling transformation can be used to move the thing being drawn.
- Viewing transformation, which maps from world coordinates to viewer coordinates. The origin of the viewer coordinate system can be thought of as the location of the viewer's "eye," and viewing transformations can be used to move the eye around in world coordinates.
- Projection transformation, which defines the boundaries of the clipping region. A projection transformation maps viewer coordinates to normalized device coordinates, and the clipping plane boundaries are at $x = +\text{-}w$, $y = +\text{-}w$, $z = +\text{-}w$. Projection transformations are used to define what region of the world is visible on the screen.
- Viewport, or NDC to DC transformation. The viewport transformation is not a full-fledged four-by-four transformation matrix; only three of the diagonal elements in the matrix can be changed. The viewport determines the mapping from normalized device coordinates to screen (device) coordinates. By default viewports are the same size as the window, although this can be adjusted.

translation. The moving of a display image in a straight line from one location to another. See **transformation**.

trimming loops. A set of oriented closed curves used to set the boundaries of a NURBS surface. See *NURBS*.

twist. A rotation around the line of sight.

unit cube. A volume defined by the following intersecting planes: $x = -1$, $x = 1$, $y = -1$, $y = 1$, $z = -1$, $z = 1$. See *normalized device coordinates*.

valuator. An input/output device that returns a value in a range. For example, a mouse is logically two valuators: the *x* position and the *y* position.

vector product. Another term for the vector cross product. If a=(a1, a2, a3) and b=(b1, b2, b3) are two 3D vectors, the vector product a times b = (a2b3-b2a3, a3b1-b3a1, a1b2-b1a2).

vertical retrace. The rate at which the monitor is refreshed. A 60 Hz monitor is redrawn 60 times per second. Same as *refresh rate*.

vertical retrace period. The amount of elapsed time between retraces of the screen. All video monitors use an electron beam to sweep the phosphors at the face of the monitor. Because the phosphors glow for only a brief period of time, the entire screen must be reswept periodically by the electron beam. On most monitors, this is done 30 times per second (30 Hz). Thus, the vertical retrace period is 1/30 second.

viewing matrix. A matrix used to describe the location of the viewer (the virtual eye looking upon a scene) in relation to the world. See *transformation*, *world coordinates*.

viewport. The mapping from normalized device coordinates to device coordinates. The viewport maps the unit cube $x/w = +\text{-}1$, $y/w = +\text{-}1$, $z/w = +\text{-}1$ to the screen space, as measured in pixels. The viewport is the last transformation in the graphics pipeline. The viewport can be smaller or larger than the window and smaller or larger than the screenmask, although in most applications, it is the same size.

window. An AIXwindows window. A rectangular area of the screen that can be moved about or placed on top of or pulled under other windows, or iconized by the user. All drawing inside the window is done by the GL process that created that window, and is totally under the control of that process. The drawing of the window borders, however, together with the window placement/iconization, is under the control of the window manager, for example, the AIXwindows window manager.

For most simple GL programs, the viewport and screenmask are set to the same size as the window. Do not confuse an AIXwindows window with the GL window subroutine, which defines a frustum in world space.

wire frame. A graphics surface-drawing technique in which the edges and contours of a primitive are represented by simple lines.

world coordinates, world space. The user-defined coordinate system in which an image is described. Modeling commands are used to position primitives in world space. Viewing and projection transformations define the mapping of the world space to screen space. See *modeling coordinates*, *screen space*, *transformations*.

writemask. A set of 8 or 12 bits (depending on the frame buffer configuration), one bit for each bitplane of the frame buffer. During any drawing operation, only those planes enabled by a 1 (one) in the bit mask can be altered. Planes set to 0 (zero) are marked read only.

z-buffer, z-buffering. Applies both to the device and the techniques commonly used as an aid in removing hidden lines and hidden surfaces. If z-buffering is enabled, each pixel will store a depth value as well as a color value. In simple terms, the depth can be thought of as the distance from the viewer's eye to the pixel. Whenever a drawing routine tries to update a pixel, it will first check the current pixel's "depth" or "z-value" and will only update that pixel with new values if the new pixel is closer than the current pixel. The region of memory that stores the z-values is also referred to as the z-buffer.

zoom factor. A multiplier to determine the amount of enlargement of a specified screen rectangle. The *x* zoom factor determines the enlargement in the *x* direction; the *y* zoom factor, in the *y* direction.

Index

T

U

V

W

Z

Readers' Comments — We'd Like to Hear from You

GL3.2 Version 4.1 for AIX: Programming Concepts (POWER-based Systems only)

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Corporation
Publications Department
Internal Zip 9561
11400 Burnet Road
Austin, TX
78758-3493

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A