IRIS® Graphics Library Programming Tools and Techniques

Document Number 007-1489-030

CONTRIBUTORS

Written by Patricia Creek and Ken Jones.
Production by Derrald Vogt.
Engineering contributions by George Kong, Sharon Fischler, Reuel Nash, David Mott, David Immel, John Rohlf, Dan Baum, Ivan Bach, Howard Cheng, Ben Garlick, Robert Keller, Martin McDonald.

IRIS® Graphics Library Programming Tools and Techniques
Document Number 007-1489-030

Contents

Examples

Figures

Tables

About This Guide

Graphics Library Programming Tools and Techniques describes Silicon Graphics® software tools and programming techniques for the graphics application developer. These tools and techniques can assist you in developing and debugging your IRIS® Graphics Library™ (IRIS GL™) application, and in analyzing and maximizing its performance.

How to Use This Guide

You'll find the information in this guide especially helpful if you:

- write IRIS GL and mixed-model programs
- use the IRIS GL Font Manager
- need assistance debugging IRIS GL programs
- require maximum performance from your IRIS GL applications
- build visual simulation applications
- build virtual reality applications
- have SkyWriter™ systems, especially multi-head systems

This guide assumes that you are familiar with the IRIS GL and that you use the IRIS GL to develop applications for Silicon Graphics IRIS-4D™ systems.

In this guide, tools are software applications that provide a graphical user interface for performing a task associated with developing a IRIS GL application. Techniques are rules, hints, and programming practices to follow for programming IRIS GL applications in a variety of situations.

What This Guide Contains

This guide contains five chapters:

- Chapter 1, “Using Fonts with the IRIS GL Font Manager,”describes how to use the IRIS GL Font Manager to provide font management for IRIS GL and mixed-model applications.
- Chapter 2, “GLX Mixed-Model Programming,” describes how to incorporate IRIS GL rendering in an X Window System™ application.
- Chapter 3, “Using GLdebug,” describes how to use GLdebug to debug IRIS GL applications. GLdebug is a graphical tool for debugging IRIS GL applications that lets you interactively control program execution, view the state of the IRIS GL while your program is running, and generate a history file.
- Chapter 4, “Tuning IRIS GL Applications,” describes how to analyze a graphics application for potential performance problems and how to improve graphics application performance by taking advantage of the pipeline architecture of IRIS-4D systems. It also outlines recommended programming practices to follow when writing application software for a variety of architectures and situations.
- Chapter 5, “Programming Visual Simulation Applications for SkyWriter Systems,” describes the special features of SkyWriter systems designed for visual simulation applications and explains how to use those features to their best advantage.
- Appendix A, “Benchmarking Tools,” contains code to assist you in taking timing measurements and provides a sample benchmark.

Related Documentation

See the *Graphics Library Programming Guide*, Volumes I and II, and the on-line Graphics Library reference (man) pages for information about the IRIS GL.

See the *Owner’s Guide* for your system for information about system architecture and operation.

See the following online books, which are viewable from the IRIS InSight™ viewer for additional information:

- *IRIX System Programming Guide*

Chapter 4	"Improving Program Performance" describes how to use IRIX profiling and optimization tools.
Chapter 7	"Using Real-Time Programming Features" describes facilities for real-time programming.

- *IRIX Advanced Site and Server Administration Guide*

Chapter 5	"Tuning System Performance" describes system tuning concepts.
Appendix A	"IRIX Kernel Tunable Parameters," describes how to tune system parameters that affect the IRIX™ operating system kernel.

Style Conventions

These typographical conventions are used in this guide:

functions()	appear in boldface font with parentheses.
arguments	appear in italic font.
file names	appear in italic font.
`code`	appears in fixed-width font.
`<key>`	appears in bold fixed-width font, surrounded by angle brackets, indicating that you press the designated key on your keyboard.
`entry`	appears in bold fixed-width font, indicating that you enter the information from your keyboard.

Chapter 1

Using Fonts with the IRIS GL Font Manager

This chapter describes how to use fonts with the IRIS GL Font Manager.

IRIS GL Font Manager Basics

The main purpose of the IRIS GL Font Manager is to make it easier to write IRIS GL and mixed-model programs that need fonts. The IRIS GL Font Manager is implemented as a library, which consists of:

- */usr/lib/libfm.so*
- */usr/lib/libfm_s*
- */usr/lib/libfm_s.a*

libfm.so is a dynamic shared object, a new type of shared library introduced in IRIX 5.0. It should be used when you develop new programs.

libfm_s is the same type of static shared library that was shipped before IRIX 5.0. It is used to run programs that were linked with a shared version of the IRIS GL Font Manager Library prior to IRIX 5.0.

libfm_s.a is a symbolic link to *libfm.so*. It is installed when you install the *gl_dev* subsystem. It is provided to avoid breaking old *Makefiles* that still have the option *-lfm_s* specified on their compile and link command lines. When you specify that option, the linker tries to link your program with */usr/lib/libfm_s.a*. It then links your program with */usr/lib/libfm.so*. In IRIX release 5.0 and later, you should use the option *-lfm* to link your program with */usr/lib/libfm.so*. The file *libfm.a* is no longer shipped because it is no longer needed.

The IRIS GL Font Manager Library provides a number of font management (fm) functions, described in "Using the IRIS GL Font Manager Library

Routines" on page 7, that can be used to get a list of the names of available font families, select and scale a specified font, draw text in a specified font, and so on.

Note: Because font management functions were written in the C programming language, they can easily be used in C and C++ programs. Fortran programs can be linked with C modules that contain calls to font management functions. ♦

If you need font management functions, you should link your program with the library */usr/lib/libfm.so* by specifying *-lfm* on the compile and link command line, such as the *cc* command line. You should also put the following statement at the beginning of each module that calls font management functions:

```
#include <fmclient.h>
```

Font Metrics

The metrics (dimensions) of a character are given in the struct *fmglyphinfo*:

```
typedef struct fmglyphinfo {
    long xsize, ysize;   /* dimensions of glyph in pixels */
    long xorig, yorig;   /* origin */
    float xmove, ymove;  /* move  */
    long gtype;          /* glyph type */
    long bitsdeep;       /* depth of pixels,in bits */
} fmglyphinfo;
```

All but two character metrics are long integers. *xmove* and *ymove* are floats. The basic unit of the values in *xmove* and *ymove* is the device unit (pixel). By making *xmove* and *ymove* floats, the IRIS GL Font Manager Library supports the subpixel positioning information needed by typesetting and graphics applications.

Figure 1-1 shows the character metrics for a pair of characters.

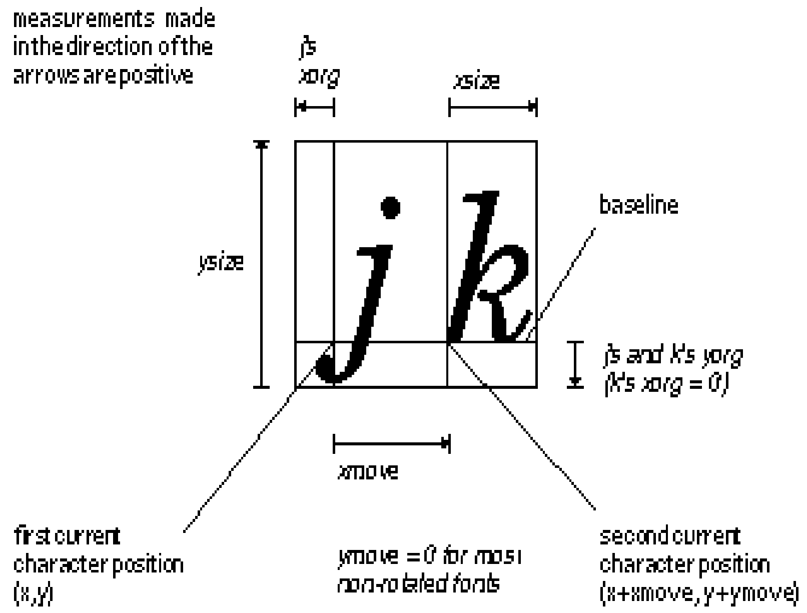

measurements made in the direction of the arrows are positive
j's xorg
xsize
ysize
j
k
baseline
j's and k's yorg (k's xorg = 0)
xmove
first current character position (x,y)
ymove = 0 for most non-rotated fonts
second current character position (x+xmove, y+ymove)

Figure 1-1 Character Metrics

A bitmap font usually contains the specification of all pixels within the bounding box of each character, without the surrounding white space. The drawing of a character starts at a reference point on the baseline. That point represents the origin of a coordinate system. The values *xorig* and *yorig* are the coordinates of the left-lower corner of the character's bounding box. The character's bitmap is drawn at those coordinates. Then you return to the reference point on the baseline, and advance to the coordinates specified by the values of *xmove* and *ymove*. That becomes the reference point for the drawing of the next character.

Note that in Figure 1-1, the values increase in the directions indicated by the arrows. Thus, *xmove* increases to the right; a negative value of *xmove* would indicate that the next character should be to the left. The value *yorig* is the distance from the bottom of the glyph to the baseline. The value *xorig* is the horizontal distance from the current character position to the left edge of the glyph; either can be a negative value. *xsize* and *ysize* are the character boundaries (for a bitmap glyph, this is the bitmap size).

Font Specification and Sizing

The API for the IRIS GL Font Manager Library is patterned after PostScript. After calling **fminit()** to initialize the IRIS GL Font Manager Library, your program should call **fmenumerate()** to find out which font families are available. Then it should call **fmfindfont()** to find one of the available font families. That function returns a handle for a 1-point font in the specified font family. If it cannot find the specified font, it returns a value of zero (0).

To get a font of certain point size, your program should call **fmscalefont()** with the handle it got from **fmfindfont()** and a point size. Your program can then call **fmsetfont()** to make the scaled font the current font, and call **fmprstr()** or **fmfprstr()** to draw a specified string of characters. Other ways of rendering text are discussed later.

Font Transformation

The IRIS GL Font Manager Library maintains an abstract notion of font rendering, called the *page*. Think of the page as a transparent sheet that is superimposed on the current window. The page maintains a coordinate system for font rendering. Application programs can make calls to the IRIS GL Font Manager Library to modify the page's transformation matrix. Changing the page's transformation matrix changes the appearance of the font in the window. You can make calls to the IRIS GL Font Manager Library if you want to render scaled or rotated text. If you do not want to alter the page's transformation matrix, you can use **fmscalefont()** to scale only the characters without scaling the page.

Conceptually, there is a distinction between scaling a font and scaling the characters of a font as they are rendered.

The code in Example 1-1 first draws a string of characters (Hello) whose height is 1 point and then draws a second string of characters (World) with a 2-point height. The text appears larger because the size of the page is doubled. The font is actually still a 1-point font, but the characters are scaled as they are rendered.

Example 1-1 Drawing Characters and Scaling the Page

```
font1 = fmfindfont("Times-Roman");
fmsetfont(font1);
fmprstr("Hello");
fmscalepagematrix(2.0);
fmprstr("World");
```

The second font drawn by the code is a true 2-point font, but the page scale is still at a 1:1 ratio. Calling **fmprstr()** draws a string of 2-point-high characters.

You can produce an identical effect using the code in Example 1-2.

Example 1-2 Scaling a Font

```
font1 = fmfindfont("Times-Roman");
fmsetfont(font1);
fmprstr("Hello");
font2 = fmscalefont(font1,2.0);
fmsetfont(font2);
fmprstr("World");
```

Example 1-2 illustrates the fact that both the font and the page have a transformation matrix. Before rendering, the font's transformation matrix is concatenated with the page's transformation matrix, and the resultant font size is rendered onto the page. The font's transformation matrix is stored with the font. The page's transformation matrix is stored in the client's process space.

To set or read the page's transformation matrix, use these routines:

```
fmconcatpagematrix()
fmgetpagematrix()
fminitpagematrix()
fmrotatepagematrix()
fmscalepagematrix()
fmsetpagematrix()
```

You can use **fmrotatepagematrix()** to rotate the page. If you then render text onto that page, a font of zero-degree rotation appears along a rotated baseline. **fmprstr()** maintains the current character position, even with rotated text.

Font Search Path

The IRIS GL Font Manager Library has a path that it searches when it looks for bitmap fonts. The default value for this font path is */usr/lib/X11/fonts/100dpi:/usr/lib/X11/fonts/75dpi:/usr/lib/X11/fonts/misc*. To override the default search value, set the environment variable *FONTPATH*. Alternatively, you can use **fmsetpath()** to load a new font path. The argument to **fmsetpath()** is a colon-separated string of directory names.

Because the IRIS GL Font Manager Library searches a path, you can store font files in different directories. During a font look-up, the IRIS GL Font Manager Library searches the directories in the order specified (left to right) by the string given to **fmsetpath()**. You can use this order to make the IRIS GL Font Manager Library use a "local" experimental font but still preserve the official font for other users.

For example, if you put the experimental font in your current directory and set *FONTPATH* to:

.:/usr/lib/X11/fonts/100dpi:/usr/lib/X11/fonts/75dpi:/usr/lib/X11/fonts/misc

the IRIS GL Font Manager Library uses the font in the current directory (dot), even if that font also exists in other specified font directories. If the IRIS GL Font Manager Library fails to find the font in the current directory, it searches for the font in other specified font directories.

The IRIS GL Font Manager Library also uses outline font files in the Type 1 font format stored in the directory:

/usr/lib/DPS/outline/base

The path for outline fonts cannot be modified.

Font Formats

This section provides references to information about font formats that are used on Silicon Graphics workstations.

About the Extended Bitmap Distribution Format Version 2.1

The Bitmap Distribution Format (BDF) was originally specified by Adobe Systems. That format was later extended for the X Window System, as described in these documents:

- *Bitmap Distribution Format 2.1* (MIT X Consortium Standard, X Version 11, Release 5). Mountain View, CA: Adobe Systems Incorporated, 1988.
- Flowers, Jim. *X Logical Font Description Conventions* (Version 1.4, MIT X Consortium Standard, X Version 11, Release 5). Cambridge, MA: Massachusetts Institute of Technology, 1989.

About the Portable Compiled Format for Bitmap Font Files

The bitmap fonts shipped by Silicon Graphics are in the PCF or compressed PCF format. Those files are produced from bitmap font files in the BDF 2.1 format. The Portable Compiled Format (PCF) format is described in Appendix D of:

- Elias, Israel and Erik Fortune. *The X Window System Server* (X Version 11, Release 5). Menlo Park: Digital Press, 1992.

About the Adobe Type 1 Font Format

The Adobe Type 1 format for outline (scalable) font files is described in:

- Adobe Systems Incorporated. *Adobe Type 1 Font Format* (Version 1.1). Reading, MA: Addison-Wesley Publishing Company, Inc., 1990.

Using the IRIS GL Font Manager Library Routines

Table 1-1 lists the font management routines in the IRIS GL Font Manager Library.

Table 1-1 IRIS GL Font Manager Library Routines

Task	**Routine**
fmconcatpagematrix()	Concatenate page matrix
fmenumerate()	List font family

Table 1-1 (continued) IRIS GL Font Manager Library Routines

Task	**Routine**
fmfindfont()	Prepare font for manipulation
fmfontpath()	Get a path for finding fonts
fmfprstr()	Render a character string in the current font without using subpixel positioning. Usually faster than fmprstr().
fmfreefont()	Free memory storage for a font
fmgetchrwidth()	Return width of a character
fmgetcomment()	Return a comment associated with a font
fmgetfontinfo()	Return overall information about font
fmgetfontname()	Return a font's name
fmgetpagematrix()	Get page matrix
fmgetstrwidth()	Return width of a string in pixels
fmgetwholemetrics()	Get information on each character in font
fminit()	Initialize the IRIS GL Font Manager Library
fminitpagematrix()	Initialize the page matrix to identity
fmmakefont()	Associate a matrix with a font
fmoutchar()	Draw a single glyph
fmprintermatch()	Toggle printer matching
fmprstr()	Draw a string in the current font with subpixel accuracy
fmrotatepagematrix()	Rotate the page
fmscalefont()	Scale a font
fmsetcachelimit()	Set the maximum cache size in quanta
fmsetfont()	Set the current font
fmsetpagematrix()	Set the page matrix
fmsetpath()	Set a path for finding fonts

The sections that follow explain how to use the routines listed in Table 1-1. These sections are organized in functional, rather than alphabetical, order.

Initializing Fonts

This section describes routines that perform various font initialization and specification functions.

Call **fminit()** to initialize the IRIS GL Font Manager Library. It sets the default page matrix for the scaling and transformation routines. You must call **fminit()** before you can make any other calls to the IRIS GL Font Manager Library routines. Its function prototype is:

```
void fminit()
```

Use **fmfindfont()** to get a font handle for a typeface. Its function prototype is:

```
fmfonthandle fmfindfont(char *face)
```

The *face* argument is a pointer to a character string that specifies a font family.

If **fmfindfont()** cannot find the font, it returns a value of zero. Otherwise, **fmfindfont()** returns a handle to a one-point high font of the specified type.

fmenumerate() accepts the name of a callback routine as an argument. It calls the specified callback routine for each font family it finds in the directories specified by the font path. **fmenumerate()** uses a string pointer to pass the name of a font family to the callback routine. Its function prototype is:

```
void fmenumerate(callback)
void (*callback)();
```

For example, the code in Example 1-3 uses **fmenumerate()** to send font names (via string pointers) to the user-defined routine, **printname()**. The **printname()** routine displays a list of available font families.

Example 1-3 Displaying a List of Available Fonts

```
void printname(str)
   char *str;
{
   printf("%s\n", str);
}

main( )
{
   fminit( );
   fmenumerate(printname);
}
```

Scaling Fonts

This section describes routines that control the size of a font.

fmscalefont() applies a scale factor to the matrix associated with the font handle passed to it, then returns a new font handle. Its function prototype is:

```
fmfonthandle fmscalefont(fmfonthandle fh, double scale)
```

Use **fmscalefont()** when you want to scale a font, but not rotate it.

fmmakefont() concatenates a matrix that is passed to it with the matrix associated with the font handle that is also passed to it, then returns a new font handle. Its function prototype is:

```
fmfonthandle fmmakefont(fh, matrix)
fmfonthandle *fh;
double matrix[3][2];
```

The transformation matrix passed in by *matrix* is multiplied with the transformation matrix in the font handle passed in by *fh*. When the font is imaged, this matrix is inspected to determine the proper scaling, shearing, rotation, or combination of these, for the imaging. This operator is more general than **fmscalefont()**, which applies uniform scaling only. For example, this multiplication can scale the font and rotate the baseline.

If you want to scale the font but do not want to rotate the baseline, it is easier to use **fmscalefont()** than **fmmakefont()**. Except for size and rotation information, the information in the new handle is copied from the handle passed in by *fh*. If the scaling of a font requests a font that does not exist, the IRIS GL Font Manager Library substitutes the closest match available.

Note: When using `matrix[3][2]`, think of it as a 2×2 transformation matrix. The last row is reserved for future development and is currently ignored. ♦

Setting the Current Font

Use **fmsetfont()** to set the current font (font handle). Its function prototype is:

```
void fmsetfont(fmfonthandle fh)
```

All subsequent rendering operations use the font handle named by *fh*. To get a font handle, use a font routine that returns a font handle, for example, **fmfindfont()**, **fmscalefont()**, or **fmmakefont()**.

Rendering Fonts

This section describes routines that render fonts to the screen.

fmoutchar() renders a single glyph, *ch*, from the current font. Its function prototype is:

```
long fmoutchar(fmoutchar fh, unsigned char ch)
```

If the glyph doesn't exist, the IRIS GL Font Manager Library advances the current character position by the width of a space. If the font does not define a space character, the IRIS GL Font Manager Library advances the current character position by the width of the font. The returned value of **fmoutchar()** is the width moved.

fmprstr() renders the characters in *str* onto the screen at the current character position, using subpixel positioning. Its function prototype is:

```
long fmprstr(char *str)
```

The font used is the one most recently named by **fmsetfont()**. The IRIS GL Font Manager Library starts rendering at the current character position and updates the current character position as it renders. Clients should use the IRIS GL **cmov()** and **getcpos()** routines to set or read the current character position.

Before calling **fmprstr()**, you must call **cmov()** to set the current character position or the results of **fmprstr()** are undefined. If the string is null, or the font does not exist, **fmprstr()** returns -1; otherwise, **fmprstr()** returns zero.

fmfprstr() renders the characters in *str* onto the screen at the current character position, without using subpixel positioning. This routine can usually render a given character string faster than **fmfprstr()**. Its function prototype is:

```
long fmfprstr(char *str)
```

The font used is the one most recently named by **fmsetfont()**. The IRIS GL Font Manager Library starts rendering at the current character position and updates the current character position as it renders. Clients should use the IRIS GL **cmov()** and *getcpos()* routines to set or read the current character position.

Before calling **fmfprstr()**, you must call **cmov()** to set the current character position or the results of **fmfprstr()** are undefined. If the string is null, or the font does not exist, **fmfprstr()** returns -1; otherwise, **fmfprstr()** returns the length of the rendered string.

Getting Font Information

This section describes routines that return information about specified fonts.

fmgetfontname() gets the name of the font associated with the font handle in *fh*. Its function prototype is:

```
long fmgetfontname(fmfonthandle fh, long slen, char *str)
```

fmgetfontname() writes information to the location pointed to by *str*. Use *slen* to tell *fmgetfontname* the size of the array pointed to by *str*. **fmgetfontname()** does not write more characters than are specified by *slen*. If there is an error in locating the font, or if no name exists for the font specified by *fh*, the returned value of **fmgetfontname()** is -1; otherwise, the returned value of **fmgetfontname()** is the length of the string actually written to *str*.

The function **fmgetcomment()** is obsolete. It has been replaced by a stub that does not do anything.

fmgetfontinfo() writes information to the members of the *fmfontinfo* type structure pointed to by the *info* parameter. Its function prototype is:

```
long fmgetfontinfo(fmfonthandle fh, fmfontinfo *info)
```

The information written to this structure pertains to the entire font that is associated with *fh*. The *fmfontinfo* data structure is defined in the */usr/include/fmclient.h* header file.

Table 1-2 lists members of the *fmfontinfo* structure that provide the most frequently used information.

Table 1-2 Members of the *fmfontinfo* Structure

Structure Member	**Meaning**
printermatched	There is a printer widths file corresponding to this font.
matrix00, *matrix01*, *matrix10*, *matrix11*	Double-precision floats that provide transformation matrix information in points.
fixed_width	All the characters in the font are the same width.
xorig, *yorig*	Coordinates of the lower-left corner of the font bounding box.
xsize and *ysize*	Maximum sizes of the characters in the font, in pixels.
height	Often the same as *ysize*, but some fonts use a larger *ysize* to get free leading (spacing between lines of text).
nglyphs	Index of the highest-numbered character. Indexing begins at 0.

Note: Some indices may not have glyphs assigned to them, but when you allocate space for **fmgetwholemetrics()**, you should use *nglyphs* + 1 as though it were the total number of characters. In other words, *nglyphs* is the highest index of a possibly sparse matrix. ♦

Getting Font Glyph Information

fmgetwholemetrics() gets glyph information associated with the font handle *fh* and writes it to the **fmglyphinfo** structures pointed to by the elements of the array *fi*. Its function prototype is:

```
long fmgetwholemetrics(fmfonthandle fh, fmglypinfo *fi)
```

You should allocate enough space to contain *nglyphs*sizeof(fmglyphinfo)*. Because **fmgetwholemetrics()** fills only those structures of the array that have corresponding glyphs in the font file, you should initialize all the *fmglyphinfo* structures before calling **fmgetwholemetrics()**. (For example, you could use **calloc()** to allocate the space. See *malloc*(3C) for more information.)

The returned function value of **fmgetwholemetrics()** is 0 if successful. If **fmgetwholemetrics()** cannot find the font referenced by the font handle, the returned function value is -1.

Getting the Width of a Character String

fmgetstrwidth() returns the number of pixels the string occupies in the *x* dimension. It uses the subpixel resolution provided in the glyph widths as it accumulates the width and rounds the sum to the nearest pixel. Rotated fonts are measured along an untransformed *x* axis. Its function prototype is:

```
long fmgetstrwidth(fmfonthandle fh, char *str)
```

Getting the Width of a Character

fmgetchrwidth() returns the number of pixels the given character occupies in the *x* dimension when it is rendered. Its function prototype is:

```
long fmgetchrwidth(fmfonthandle fh, unsigned char ch)
```

The returned value is rounded to an integer. If that character glyph does not exist, the width of a space is returned. If a space does not exist, the width of the font is returned. Rotated fonts are measured along an untransformed *x* axis.

Getting and Setting the Font Environment Variables

This section describes routines that affect the environment in which fonts are managed.

Getting the Font Search Path

fmfontpath() returns a pointer to a string that describes the current search path for finding font files. Its function prototype is:

```
char *fmfontpath()
```

The path is a colon-separated list of directories that originate at the root. The default path is */usr/lib/fmfonts*. To reset the value of the font path, use **fmsetpath()**.

Setting the Font Search Path

fmsetpath() accepts a pointer to a string that describes the current search path for finding font files. Its function prototype is:

```
void *fmsetpath(char *path)
```

The path is a colon-separated list of directories that originate at the root. The default path is */usr/lib/fmfonts*.

Managing Font Memory

Memory management for the IRIS GL Font Manager Library is under user control.

Note: Previous releases of the IRIS GL Font Manager Library used caching to restrict its use of memory for fonts. For compatibility, the font-caching routines are still in the library as stubs, but they are not functional. ♦

fmfreefont() frees the storage associated with a font in a given rotation and size, specified by the font handle *fh*. Its function prototype is:

```
void fmfreefont(fmfonthandle fh)
```

Freeing a font also frees the font handle. To ensure that **fmfreefont()** frees the correct font/rotation/size instance, be sure that the same page matrix is in force as when you first queried or rendered from that font.

Because normal usage of the IRIS GL Font Manager Library does not involve changing the page matrix, you seldom need to worry about it. But if you find that you cannot delete a font (or have deleted the wrong font) you may have rotated the page matrix. One way to avoid this problem is to call **fmfreefont()** only when the page matrix is not rotated. Rotated fonts are created and destroyed as necessary and do not need explicit deletion.

Adjusting Widths to Match Laser Printers

Many applications render text on the screen to give the user the chance to proof the text before printing it on a laser printer. For a more realistic simulation, you should use laser printer character widths to represent the text.

fmprintermatch() sets a state variable that controls printer font matching. Its function prototype is:

```
void fmprintermatch(long set)
```

Call:

fmprintermatch(1)	to enable printer matching
fmprintermatch(0)	to disable printer matching

When the IRIS GL Font Manager Library renders (images) a font, it inspects the state of this variable. If enabled, the IRIS GL Font Manager Library searches for a printer widths file that corresponds to the font. If the file exists, and the font has not yet been sized, the IRIS GL Font Manager Library creates a new font. The IRIS GL Font Manager Library also updates the font handle of the current font so that it has character widths that correspond to the laser printer’s width scheme.

Transforming the Page

The page transformation is stated in the page matrix. This section describes routines that let you inspect and change the state of the page matrix.

Note: When using matrix[3][2], think of it as a 2×2 transformation matrix. The last row is reserved for future development and is currently ignored.

fminitpagematrix() initializes the page matrix to an orthographic projection. Its function prototype is:

```
void fminitpagematrix()
```

fmsetpagematrix() loads the page matrix verbatim with matrix *mat*. Its function prototype is:

```
void fmsetpagematrix(mat)
double mat[3][2];
```

fmgetpagematrix() returns the page matrix in *mat*. Its function prototype is:

```
void fmgetpagematrix(mat)
double mat[3][2];
```

fmscalepagematrix() uniformly scales the page matrix by *scale*. Its function prototype is:

```
void fmscalepagematrix(double scale)
```

fmrotatepagematrix() post-concatenates a rotation to the page matrix, where the rotation is measured in a counter-clockwise direction in degrees. Its function prototype is:

```
fmrotatepagematrix(double angle)
```

You can also use **fmrotatepagematrix()** to generate a screen font that is exactly (within one pixel) the specified size. You should try this in a test program first to see whether the possible degradation in quality is acceptable. This “roughness” comes from the need to scale a bitmap font if that font does not exist at the specified size.

For example, the IRIS GL Font Manager Library normally renders text using a bitmap font that is the closest match possible to the requested size. But, if you rotate the page matrix, even by one 1/1000 of a degree, the IRIS GL Font

Manager Library tries to create a font that is rotated that much. As a side effect, the IRIS GL Font Manager Library also distorts (shrinks or stretches) the page to generate a font that is within a pixel of the specified size; however, stretching or shrinking a bitmap often results in "rough" looking characters.

To try scaling a bitmap, call:

```
fmrotatepagematrix(.01)
```

then print a string with **fmprstr()**.

fmconcatpagematrix() post-concatenates the page matrix with *mat*. Its function prototype is:

```
void fmconcatpagematrix(mat)
double mat[3][2];
```

IRIS GL Font Manager Library Example Program

The program in Example 1-4 writes a string of green, 25-point characters to a window, beginning at window coordinate (30, 100). Compile the program using the following command line options:

```
cc example.c -o example -lc_s -lfm_s -lgl_s
```

Example 1-4 Using IRIS GL Font Manager Routines in an IRIS GL Program

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

main( )
{
    short val;
    fmfonthandle font1, font25;

    prefsize(240,210);
    winopen("Hello");
    color(BLACK);
    clear( );
    color(GREEN);
    fminit( );
    /* Exit if can’t find the font family */
    if ((font1=fmfindfont("Times-Roman")) == 0) exit (1);
    /* scale the 1-point-high font to 25 points */
    font25 = fmscalefont(font1, 25.0);
    fmsetfont(font25);
    cmov2i(30, 100);
    fmprstr("Hello World!");
    while(TRUE) { /* redraw window if necessary */
          if (qread(&val) == REDRAW) {
              reshapeviewport( );
              color(BLACK);
              clear( );
              color(GREEN);
              cmov2i(30, 100);
              fmprstr("Hello World!");
          }
    }
}
```

Remote Font Management

Remote font usage, like the Network Transparent IRIS GL, follows the X remote font model.

The old shared IRIS GL Font Manager Library (*/usr/lib/libfm_s*) and the new shared IRIS GL Font Manager Library (*/usr/lib/libfm.so*) work with either local or remote graphics service, or both within the same application.

All IRIS GL Font Manager Library routines are rerouted at their calling point. This means that when the remote graphics routine is active, all calls are executed on the remote host. Because of this, a font directory containing the needed font data must reside on the remote host—font data is not transmitted over a DGL socket connection. Only high-level calls to render and manipulate text are transmitted over the socket connection.

For applications that open both local and remote windows, the user must note that the local and remote IRIS GL Font Manager Library services are disjoint. The user must note which server is currently active by following the rules for the network-transparent IRIS GL. To summarize how the current server is determined:

1. After **dglopen()** has been called, the current server is that specified by the **dglopen()** call. This remains in effect until the next **dglopen** or **winset()** call.
2. In the absence of a **dglopen()** call, the current server is determined by an environment variable such as *DISPLAY* and the defaulting mechanism.

To use the IRIS GL Font Manager Library in a multi-server environment:

1. **fminit()** must be called once for each graphics server on which IRIS GL Font Manager Library facilities are desired. Only one call to **fminit()** is needed, regardless of the number of windows you want to use on the specified server.
2. A font handle returned by **findfont()** is usable only in the windows controlled by the server from which the request originated; therefore, the user must remember which server originated the **findfont()** request.
3. Matrix operations pertain only to the currently active server.

Chapter 2

GLX Mixed-Model Programming

Most X-based applications are limited to 2D graphics. You can use the IRIS GL to add one or more 3D rendering windows to an X application. For example, you can manipulate a 3D IRIS GL image using an IRIS IM[1] control panel. Using one or more IRIS GL windows in an X application, mixing X and IRIS GL, is called mixed-model, or GLX, programming.

"IRIS Graphics Library Programming in the X Environment" on page 22 highlights some considerations for IRIS GL programming in the X environment.

When writing a mixed-model program, you have two choices: you can use the Xt toolkit and a widget set such as IRIS IM, or you can write your program in Xlib and IRIS GL using special GLX commands. The first mixed-model programming method, using Xt, is much easier to use and is more commonly used by mixed-model developers. Silicon Graphics provides a widget library that simplifies mixed-model programming with Xt. "Using IRIS GL Widgets to Create Mixed-Model Programs" on page 31 explains how to write a mixed-model program using Xt, the IRIS Widget Library, and the IRIS GL widget, GlxDraw (Silicon Graphics also provides an IRIS IM version of GlxDraw, called GlxMDraw). Sample programs in "Mixed-Model Sample Programs Using Widgets and Xt" on page 45 demonstrate these concepts.

If you prefer to create a mixed-model program in Xlib, without using Xt, refer to the recommended references on X programming, and use the four mixed-model routines described in "Using Xlib to Write a GLX Program" on page 72. The sample program in "Mixed-Model Example Program Using Xlib and IRIS GL" on page 73 at the end of "Using Xlib to Write a GLX Program" contains an example of a mixed-model program created with Xlib.

[1] IRIS IM is Silicon Graphics' port of the industry-standard OSF/Motif™.

The mixed-model programming routines documented in this guide are subject to change when the next version of the Graphics Library is released.

IRIS Graphics Library Programming in the X Environment

This section provides some general information about how to write an IRIS GL program in the X environment. In this guide, a pure IRIS GL program is one that does not include X—a pure IRIS GL program uses IRIS GL calls even for tasks such as input handling, which are governed by the X server.

Silicon Graphics recommends that you write *mixed-model* programs rather than pure IRIS GL programs. A mixed-model program is essentially an X program that uses the IRIS GL to handle graphics. In a mixed-model program, the IRIS GL is completely removed from all areas governed by the X server.

Using Network Transparency

The IRIS Graphics Library is network transparent. Network transparency allows an application running on one host to display on a remote host. Any IRIS GL program can follow a display selection scheme based on the X Windows *DISPLAY* variable.

By setting the *DISPLAY* environment variable, you can render on the screen of another workstation. The IRIS GL follows the X display syntax, reading the *DISPLAY* environment variable to determine the correct screen for rendering. If the *DISPLAY* environment variable is not set, the IRIS GL checks the other environment variables until it determines the proper location.

The full X display syntax supported is:

```
[[userid@]hostname [#port]]:server[.screen]
```

Using the Shared IRIS GL

There are two IRIS GL libraries: shared (*libgl_s.a*) and nonshared (*libgl.a*). Shared libraries provide optimum use of system resources and the best portability and compatibility between platforms. The shared IRIS GL is network transparent. To use network-transparent features, you must link with the shared IRIS GL using *-lgl_s*.

The nonshared graphics library (*libgl.a*) is provided for use with some debugging tools. IRIS GL programs using the nonshared library cannot use network-transparent features, nor can you use *gldebug* on them.

Flushing Buffered Graphics Data

Because IRIS GL programs are network transparent, you must use **gflush()** to ensure that buffered graphics data is transferred. A single-buffered program that never swaps buffers or never reads the event queue may not ever display any graphics without **gflush()**.

A **gflush()** is built in to some IRIS GL commands, such as **qread()** and **swapbuffers()**, but these routines are illegal in mixed-model programs.

In mixed-model programs, you may need to put **gflush()** at the end of all callbacks and actions. Alternatively, if there is a single routine that draws the scene each time, you can put **gflush()** at the end of that function. You should be careful about where you place **gflush()**, because it can adversely affect graphics performance if you place it inside a loop or other time-critical location. For more information about network-transparent features, see *gflush*(3G) and the *Graphics Library Programming Guide*.

Using Cursors

X maintains a cursor color and shape for each window. Pure IRIS GL programs do not need to re-color the cursor when they get input focus. It is not possible to call **mapcolor()** on the cursor colormap to change the colors for all cursors.

Handling Input Events

This section describes the two different ways of handling user input, and it provides information about how the X server relates to input events.

Differences Between Pure IRIS GL and Mixed-Model Event Handling

When your IRIS GL program calls an IRIS GL windowing routine, the IRIS GL windowing routine passes information directly to the X server. In turn, the X server puts events in the IRIS GL event queue so that existing IRIS GL programs can monitor and react to events. Thus, IRIS GL events originate from the X server and IRIS GL event calls such as **qenter()** are implemented using a combination of X protocol requests to the X server. When **qread()** is called to read the event queue, X events are received from the X server and translated into the appropriate IRIS GL events.

The IRIS GL queue size is given by GL_MAXQ in */usr/include/gl/qcontrol.h*, which is currently defined as 600.

Using the IRIS GL windowing routines precludes the use of any of the X event management routines, any X-based toolkit, and any Graphical User Interface (GUI) widgets that such a toolkit may define.

Currently, the IRIS GL interface does not include a library of GUI widgets, although it does contain routines that support the creation of popup menus. If you want to use a dialogue box or some other GUI widget (except for menus), you must write the widget code from scratch.

If you must create a GUI widget library from scratch or add to an existing library, you should consider buying the Showcase™ source code and using its user-interface code for dialog boxes, alert boxes, slide bars, and buttons.

Although an IRIS GL program cannot call X-based widgets, an X program can create a subwindow that uses the IRIS GL for rendering. Such an X program is called a mixed-model program. In a mixed-model program, you can use the widgets of an X-based toolkit (such as IRIS IM) to handle the user interface.

In a mixed-model program, you cannot use any of the IRIS GL event or windowing management routines such as **winopen()** or **qread()**, or any of the popup menu routines such as **dopup()**. In a mixed-model program, the X part of the code must manage all of the event handling, window control, and menus.

Filtering the Number of Mouse Input Events

Under X, the mouse generates about 100 events per second. You can set an IRIS GL compatibility resource, *glCompat.motionQGrowthRate*, to filter the number of events to a more appropriate scale for applications that do a lot of mouse processing. You can use **glcompat()** to set the number of mouse input events to a rate compatible with IRIX system software releases prior to 4D1-4.0, or to provide maximum compression of mouse motion events.

For compatibility, use:

```
glcompat(GLC_MQUEUERATE, GLC_COMPATRATE)
```

For maximum compression, use:

```
glcompat(GLC_MQUEUERATE, GLC_CMPRESS)
```

Event Buffering

When an IRIS GL call gets an input event, the X server is queried for the event. All outstanding events are read from the X server, and those events are stored in an event buffer in *libgl*. If a user moves the mouse, generating 10 events, then the next **qread()** call done by the program causes the last 9 of those events to be stored in a *libgl* array in memory. The first event is returned by **qread()**.

Using select() on the IRIS GL File Descriptor

As a result of the way *libgl* buffers events, programs that use the IRIX call **select()** on the IRIS GL file descriptor returned by **qgetfd()** should call **gflush()** before calling **select()**, to flush rendering before waiting for input. The application should also empty out the IRIS GL queue after returning from **select()**.

Two concepts are important when using **select()**:

- It is important to consume all IRIS GL events that are waiting in the queue before returning to the **select()** in order to maintain proper input behavior.
- IRIS GL timers do not work with the **select()** mechanism. You should modify the **select()** timeout to adjust time intervals rather than using IRIS GL timers to regularize event scheduling.

The sample code below demonstrates the proper use of **select()**:

```
gl_fd = qgetfd();
FD_SET(gl_fd &fds);
while (1) {
   gflush();    /* Flush rendering before checking for input
   */
   select(...);
   if (FD_ISSET (gl_fd, &fds)) {
      while (qtest()) {
           short val;
           Device dev = qread(&val);
            ...   /* process the event */
}
```

If the program were to execute only a single **qread()** after returning from **select()**, instead of using the loop structure shown in the sample code, this sequence of events could occur:

1. The user moves the mouse, generating 10 mouse-motion events.
2. In the IRIS GL program, **select()** returns, indicating that there is an IRIS GL event to read.
3. The program does a **qread()**, which returns the first of the 10 events and stores the remaining events in an internal *libgl* buffer. As far as the X server is concerned, it has returned all 10 events to the client.
4. The next **select()** hangs forever.

Windowing

You can handle windowing with IRIS GL windowing routines, such as **winopen()**. A **winopen()** call provides results similar to the X calls **XOpenDisplay()** and **XCreateWindow()**. These X events are then translated into IRIS GL events.

Note: Where possible, it is much better to write mixed-model programs and handle windowing with X calls, rather than IRIS GL calls. ♦

Backing Store and Save Under

In the X Window System environment, when an X window is covered by another window, the contents of the covered window may be saved. The pixels saved by the X server are known as *backing store*. Even though the X server may save and restore the contents of obscured windows, an application must always be prepared to redraw itself in response to X *Expose* events. This is because the X server is always free to ignore requests for backing store, and it may discard existing backing store at any time if memory runs out.

Backing store requests are ignored for all IRIS GL windows. Because IRIS GL rendering bypasses the X server and goes directly to the hardware, drawing requests for obscured areas cannot be trapped by the X server and are subsequently redirected to backing store. As a result, whenever an IRIS GL window is exposed, it will receive a redraw event and it should redraw itself. Use of overlay planes may help to reduce the number of redraws required for IRIS GL windows.

Application programmers should not automatically assume that use of backing store is faster than redrawing damaged windows. On some Silicon Graphics workstations, redrawing a window is faster than saving and restoring pixels. Backing store can also use large amounts of memory. For example, backing store for a full-screen-sized window that is 24 bits per pixel can use as much as 5 megabytes of memory. Backing store for several large 24-bit windows can easily use up all memory and cause excessive memory thrashing.

For more information on backing store, see O'Reilly Vol. 1, Section 4.3.5, "Backing Store."

Save Under is another example of the X Window System storing pixels for windows that are covered by other windows. Save Under is another action that you may not be able to rely upon, because it is not supported for IRIS GL windows.

Writing Mixed-Model Programs

A mixed-model program is an X program. It allows full access to the capabilities of X by completely removing the IRIS GL from any feature governed by the X server. X gives the programmer direct control of all the areas governed by the X server. You can't create mixed-model programs that go only halfway. Your mixed-model program *must* use X for all window-system-related services.

You can find examples of many mixed-model programs in the *4Dgifts* directories. If you have trouble finding the relevant directories, refer to the *README* file in */usr/people/4Dgifts*. This file explains the contents and organization of the *4Dgifts* directories.

The Difference Between Mixed-Model Programs and Multi-Client Programs

Under previous releases of the IRIX operating system, it was possible for a single program to open some windows with the IRIS GL, open some windows with X, and draw into each kind of window with the appropriate library. This is one program that is both an X and an IRIS GL client, not a mixed-model program. It is important to understand the difference between a mixed-model program and this type of program, which is sometimes called a *multi-client* or a *split-model* program. In a mixed-model program, IRIS GL windows can be nested within one another. In a split-model program, IRIS GL and X interfaces remain in separate top-level windows. The interaction between X and IRIS GL windows in split-model programs is not as easy to implement nor is it as elegant and complete as the interaction provided by mixed-model programs.

Network Transparency and gflush()

Mixed-model programs are network transparent; that is, simply by setting the DISPLAY environment variable, the same binary can be made to image locally or remotely on a machine supporting the network-transparent IRIS GL.

To ensure transfer of buffered graphics data across the network, mixed-model programs must call **gflush()**, as must pure IRIS GL programs. A good place to call **gflush()** is before waiting for user input. In a widget-based program, it is a good idea to call **gflush()** at the end of every callback that performs IRIS GL drawing. For more information on how to use **gflush()**, see the *Graphics Library Programming Guide*.

X Rendering is Not Possible in an IRIS GL Window

Once the IRIS GL has been bound to a window, X rendering in that window yields undefined results. Subwindows may exist or can be created, in which rendering works correctly.

Incompatible IRIS Graphics Library Calls

Most IRIS GL code that follows the features documented in the *Graphics Library Programming Guide* can be used in mixed-model programs, but a few IRIS GL functions are not compatible with mixed-model programming. In mixed-model programs, facilities managed by the X server are accessed through X, rather than through the IRIS GL. It is illegal to call any IRIS GL routine that accesses window-system-controlled features from within a mixed-model window.

These routines are illegal, and the areas they govern must be handled through X in a mixed-model program:

cursors	**attachcursor()**, **curorigin()**, **curstype()**, **defcursor()**, **getcursor()**, **RGBcursor()**, **setcursor()**
colormaps	**blink()**, **cyclemap()**, **getcmmode()**, **getmap()**, **getmcolor()**, **mapcolor()**, **multimap()**, **onemap()**, **setmap()**
device input	**getbutton()**, **getvaluator()**, **noise()**, **qdevice()**, **qread()**, **qtest()**, **tie()**, **unqdevice()**
device control	**qcontrol()**, **qenter()**, **setvaluator()**
display mode	**acsize()**, **drawmode()**, **overlay()**, **underlay()**
framebuffers	**cmode()**, **gconfig()**, **doublebuffer()**, **leftbuffer()**, **rightbuffer()**, **singlebuffer()**, **stereobuffer()**, **RGBmode()**

windows	**foreground()**, **fudge()**, **keepaspect()**, **minsize()**, **maxsize()**, **noport()**, **noborder()**, **prefsize()**, **prefposition()**, **stepunit()**, **screenspace()**, **winconstraints()**, **winget()**, **winmove()**, **winopen()**, **winpush()**, **winpop()**, **winposition()**, **winset()**, **wintitle()**

The window-shaping routines that follow work in mixed-model programs, but they are not recommended. You should obtain this information from the window system if possible.

window shape	**getorigin()**, **getviewport()**, **reshapeviewport()**

In a mixed-model program, instead of using these incompatible routines, you can use Xt, GlxDraw, and the four special mixed-model calls described in "Using the GLX Mixed-Model Routines" on page 72.

In mixed-model programs, window depth and display mode are window attributes that are defined when the window is created, and they cannot be changed. To change these attributes, you must create a new window. If you need multiple display modes in your application, you can create multiple windows, then map and unmap them, or raise one above the others.

Installing Colormaps

When using colormaps in GLX programs, it is a mistake to not call **XSetWMColormapWindows()**. A mixed-model program must use **XSetWMColormapWindows()** to make sure its colormaps are installed. Even if a program is using RGB mode, **XSetWMColormapWindows()** should still be called because some hardware (such as IRIS Indigo) simulates RGB with a colormap.

If you don't call **XSetWMColormapWindows()**, the default X colormap is used. This may not cause any obvious problems on single colormap systems such as the Personal IRIS, but it may cause problems on multiple colormap systems such as the IRIS Indigo. For example, on an IRIS Indigo Entry system, which simulates RGB with a colormap, this means that the IRIS GL colormap does not get installed, so the resulting colors are incorrect. It is a good idea to test programs on both types of systems.

See the *cmapov.c* sample program at the end of this chapter for a demonstration of installing a custom colormap. For more general information on colormaps, see the *IRIS IM Programming Notes*.

Using IRIS GL Widgets to Create Mixed-Model Programs

The addition of direct control over X features makes mixed-model programs more complex than pure IRIS GL programs. In general, you can bypass many of the complexities of X and of mixed-model programming by using the Xt toolkit and a widget set such as IRIS IM.

When mixing the IRIS GL with Xt, IRIS IM, or Athena widgets, you can use the Silicon Graphics mixed-model GlxDraw widget, which simplifies mixed-model programming with IRIS IM or any other widget set. The GlxDraw widget is also compatible with User Interface Language (UIL). This section explains how to use the GlxDraw widget for embedding IRIS GL in an Xt or IRIS IM program.

This section also discusses overlays and some Xt features that are useful in mixed-model programming, such as the features for handling input and dealing with animation that are discussed in “Animation: Timeouts and Workprocs” on page 45.

What You Need to Know About Xt and IRIS IM

The examples shown in this section use Xt and IRIS IM. Although knowledge of Xt and IRIS IM is not required to read this section, understanding the details of the examples does require some Xt and IRIS IM knowledge. This chapter points out areas of the Xt and IRIS IM toolkits that are of special interest to mixed-model programmers—it does not provide a tutorial on Xt and IRIS IM. For more information on the relevant features of Xt and IRIS IM consult the OSF/Motif series, and Digital’s *X Window System Toolkit: the Complete Programmer’s Guide and Specification*, or O’Reilly’s Vols. 4 & 5 on X Toolkit Intrinsics.

You Don't Have to Use IRIS IM

This section refers frequently to IRIS IM because it is commonly used in mixed-model programs; however, unless otherwise specified, you can use the features discussed here with other widget sets, such as the Athena widget set because the features discussed in this chapter exist either within the widget itself or are based on the X toolkit. If you do use IRIS IM, you should use GlxMDraw, the IRIS IM version of the GlxDraw widget.

About the GlxDraw and GlxMDraw Widgets

Use the GlxDraw widget when creating a mixed-model program using Xt. The GlxDraw widget is similar to a normal widget, but it sets up a configuration for IRIS GL drawing, as well as providing resources and callbacks that are useful to the IRIS GL programmer. The GlxDraw widget also provides support for overlays.

There are actually two GlxDraw widgets. The widget known as GlxDraw is a generic widget, suitable with any widget set that is based on the Xt intrinsics. There is also a version known as GlxMDraw (note the M) for use with IRIS IM programs.

The two widgets are very similar, but they do have these differences:

- GlxMDraw is a subclass of the IRIS IM XmPrimitive rather than being a subclass of the Xt Core widget and, therefore, has various defaults such as background color.
- GlxMDraw understands IRIS IM traversal, although traversal is turned off by default.
- GlxMDraw has an IRIS IM style creation function, **GlxCreateMDraw()**, in addition to allowing creation of the widget directly through Xt.

In all other respects, the two widgets are identical. The remainder of this chapter refers to the GlxDraw widget, but unless otherwise specified, everything stated refers to both.

Using GlxDraw

Follow these basic steps for writing a mixed-model program using Xt and the GlxDraw (or GlxMDraw) widget:

1. Include the appropriate header file.
2. Declare the GlxConfig resource to describe the IRIS GL requirements.
3. Create the GlxDraw widget (include the GlxConfig resource).
4. Add callbacks and provide callback routines.
5. Write the IRIS GL code.
6. Handle the input events.
7. Link with the IRIS Widget Library.

A sample program that demonstrates these concepts follows, and the sections that follow the sample program describe each step in detail, except steps 5 and 6, which are covered more thoroughly in “Handling Input in a Mixed-Model Program” on page 42.

A Sample Program Using GlxDraw

The sample program in Example 2-1 uses the GlxDraw widget. Details about the code are discussed in the sections that follow.

Example 2-1 An Example of Using the GlxDraw Widget

```
#include <X11/Xirisw/GlxDraw.h>
. . .
/* The following configuration should match your
 * hardware needs as described in GLXgetconfig
 */

GLXconfig glxConfig [] = {
   { GLX_NORMAL, GLX_DOUBLE, TRUE },
   { GLX_NORMAL, GLX_RGB, TRUE },
   { GLX_NORMAL, GLX_ZSIZE, GLX_NOCONFIG },
   { 0, 0, 0 }
};
```

```
main()
{
    Arg args[10];
    int n;
    Widget parent;   /* The parent of the gl widget */
    Widget glw;      /* The glxDraw widget          */
    . . .
    /* Create the widget */
    n = 0;
    XtSetArg(args[n], GlxNglxConfig, glxConfig); n++;
    glw = XtCreateManagedWidget(“glx”, glxDrawWidgetClass,
                                parent, args, n);
    /* Add any needed callbacks */
    XtAddCallback(glw, GlxNginitCallback, ginitCB, 0);
    XtAddCallback(glw, GlxNexposeCallback, exposeCB, 0);
    XtAddCallback(glw, GlxNresizeCallback, resizeCB, 0);
    /* Also add input callback if needed */
    . . .
    XtRealizeWidget(toplevel);
    /* install the colormap after the widget is realized */
    installColormap (toplevel, glw);
}
/* The initialize callback */
static void
ginitCB(w, client_data, call_data)
    Widget w;
    caddr_t client_data;
    GlxDrawCallbackStruct *call_data;
{
    GLXwinset(display, call_data->window);
    /* Perform any necessary graphics initialization.*/
}

/* a function to install the colormaps */
installColormap(toplevel, glw)
Widget toplevel, glw;
{
    Window windows[2];
    windows[0] = XtWindow(glw);
    windows[1] = XtWindow(toplevel);
    XSetWMColormapWindows(XtDisplay(toplevel),
                          XtWindow(toplevel),windows, 2);
}
```

```
/* The expose callback */
static void
exposeCB(w, client_data, call_data)
    Widget w;
    caddr_t client_data;
    GlxDrawCallbackStruct *call_data;
{
    GLXwinset(display, call_data->window);
    draw_scene();  /* User provided routine to redraw */
}

/* The resize callback */
static void
resizeCB(w, client_data, call_data)
    Widget w;
    caddr_t client_data;
    GlxDrawCallbackStruct *call_data;
{
    GLXwinset(display, call_data->window);
    viewport(0, (Screencoord) call_data->width-1,
             0, (Screencoord) call_data->height-1);
}
```

Including GlxDraw Header Files

The header file to include depends on whether the IRIS IM or the generic version of the program is included.

For the IRIS IM version:

```
#include <X11/Xirisw/GlxMDraw.h>
```

For the generic version:

```
#include <X11/Xirisw/GlxDraw.h>
```

Declaring the GLXconfig Resource

In mixed-model programs, you must configure your windows before you can render in them, rather than configuring them on the fly as you can in pure IRIS GL programs—you cannot use the IRIS GL routine **gconfig()** in mixed-model programs.

The GLXconfig resource takes an array describing IRIS GL requirements such as single or double buffering, RGB or color index mode, z-buffering, accumulation buffering, overlay/underlay/popup windows, and so on. GLXconfig determines what is possible on the current hardware and returns a new structure with the exact description of what will be allowed, as well as the information needed to create an X window suitable for IRIS GL drawing. You can use **GLXgetconfig()** to return the actual configuration. See the *GLXgetconfig*(3G) manual page for more information.

The structure prototype is:

```
typedef struct _GLXconfig {   /* from <gl/glws.h> */
    int   buffer;
    int   mode;
    int   arg;
} GLXconfig;
```

Values for *GLXconfig.buffer* specify which framebuffer the configuration affects. Possible values are *GLX_NORMAL*, *GLX_POPUP*, *GLX_OVERLAY* and *GLX_UNDERLAY*.

Values for *GLXconfig.mode* specify which attributes of the buffer are being configured. The interpretation of *GLXconfig.arg* is dependent on the mode that is specified.

Values for the *mode* and *arg* fields of the GLXconfig are:

GLX_DOUBLE

In the input configuration, single buffering is assumed unless *GLX_DOUBLE, True* is specified. On output, if double buffering is not available, the *arg* field for *GLX_DOUBLE* will be false.

GLX_RGB

In the input configuration, color index is assumed unless *GLX_RGB, True* is specified. On output, if RGB is not available, the *arg* field for *GLX_RGB* will be false.

GLX_BUFSIZE

If not specified, or if the *arg* is *GLX_NOCONFIG*, then the largest available number will be allocated. On return, the *arg* field will contain the number allocated.

GLX_STENSIZE, *GLX_ACSIZE*, *GLX_ZSIZE*
If none of these buffers are specified, none will be allocated. If the *arg* is *GLX_NOCONFIG*, the largest available number will be allocated. On return, the *arg* field will contain the number allocated.

GLX_VISUAL Ignored on input. On output, it contains the correct visual ID for the window to be created for that buffer.

GLX_COLORMAP
Ignored on input. In the output, the value is the colormap that traditional **winopen()**-style IRIS GL windows will use for that buffer. This is for information only—colors in this colormap can be queried, and this colormap ID can be used in creating windows. However, it's forbidden to write into this colormap using **XStoreColor**(s). If a client wants to do **XStoreColor**(s), it needs to create its own colormap. Even if a client does not do **XStoreColor**(s), it is free to create windows with a colormap other than the one returned in *GLX_COLORMAP*.

GLX_WINDOW
Ignored on input. On output, for a supported buffer request, it is a placeholder with value *GLX_NONE*. Before the output buffer is passed to **GLXlink()**, the value should be replaced with the window that was created.

GLX_MSSAMPLE
On input, the *arg* field should contain the requested number of multisample samples to be stored at each framebuffer pixel location. On output, the *arg* will contain the allocated number of samples. If not specified, no samples will be allocated.

GLX_MSZSIZE
On input, the *arg* field should contain the requested number of bits per depth component desired in the multisample buffer. On output, the *arg* will contain the allocated number of bits. If none are specified, no bits will be allocated.

GLX_MSSSIZE
On input, the *arg* field should contain the requested number of bits per stencil field desired in the multisample buffer. On output, the *arg* will contain the allocated number of bits. If none are specified, no bits will be allocated.

GLX_STEREOBUF
In the input configuration, monoscopic buffer is assumed unless *GLX_STEREOBUF, True* is specified. On output, if stereoscopic viewing is not available, the *arg* field for *GLX_STEREOBUF* will be *false*.

The following code fragment, reproduced from the sample program, requests double buffered RGB mode with z-buffering:

```
GLXconfig glxConfig [] = {
    { GLX_NORMAL, GLX_DOUBLE, TRUE },
    { GLX_NORMAL, GLX_RGB, TRUE },
    { GLX_NORMAL, GLX_ZSIZE, GLX_NOCONFIG },
    { 0, 0, 0 }
};
```

Creating the GlxDraw Widget

Create the GlxDraw widget, just like any other widget, as part of the widget hierarchy. IRIS IM users may wish to create it as the child of a frame, as it provides no decoration of its own. You can create the IRIS IM version of the widget by using the IRIS IM style convenience function:

```
GlxCreateMDraw(parent,name,arglist,argcount)
```

Alternately, you can create the IRIS IM version of the widget directly, using Xt creation functions:

```
XtCreateWidget(name,glxMDrawWidgetClass,parent,arglist,argcount)
```

To create the generic version of the widget, you must use an Xt Creation function, such as:

```
XtCreateWidget(name,glxDrawWidgetClass,parent,arglist,argcount)
```

The sample program in Example 2-1 creates a generic GlxDraw widget using the Xt creation function.

The following code fragment creates an IRIS IM style widget using the convenience function:

```
main(argc, argv)
int argc;
char *argv[];
{
    Arg args[20];
    int n;
    Widget parent, glw;
    . . .
    n = 0;
    XtSetArg(args[n], GlxNglxConfig, glxConfig); n++;
    glw = GlxCreateMDraw(frame, "glwidget", args, n);
    XtManageChild (glw);
    . . .
}
```

Inserting Callbacks

Xt programs handle events through a callback mechanism, whereby the user provides functions to be called when certain events occur. The GlxDraw widget provides several callbacks, which are listed in Table 2-1.

Table 2-1 GlxDraw Callbacks

Callback	Description
ginit	This callback is the first callback. It is called automatically when a widget is realized and the window is created.
expose	This callback is called whenever the window needs redrawing, for example if an overlapping window is removed.
resize	This callback is called whenever the window is resized.
input	This callback is called for keyboard and mouse input.
overlayExpose	This callback is called whenever the overlay window needs redrawing. There are similar callbacks for underlays and popups.

After creating the widget, add the necessary callbacks with **XtAddCallback()**, as demonstrated in this code fragment from the sample program:

```
XtAddCallback(glw, GlxNginitCallback, initCB, 0);
XtAddCallback(glw, GlxNexposeCallback, exposeCB, 0);
XtAddCallback(glw, GlxNresizeCallback, resizeCB, 0);
```

The *ginit* callback is needed because Xt doesn't create windows immediately. Instead, it waits until the widget is realized (usually by a call to **XtRealizeWidget()**). You can't use the IRIS GL in a window until it is realized. The application can either perform its IRIS GL initialization after realizing the widget, or it can use the *ginit* callback to perform this function.

Here are some things *not* to do within the *ginit* callback:

- Do not call **winopen()**. The widget has already created the window. **winopen()** is not allowed in mixed-model programs.
- Do not call **gconfig()**. This is not allowed in mixed-model programs. Use the **GlxConfig()** resource to set up IRIS GL configuration.
- Do not draw the window. After the window is created, it gets an expose callback (unless it is hidden). Drawing the window during initialization is unnecessary.

The second callback states that when the widget *glw* receives an *Expose* callback, the user provided routine *exposeCB* should be called, with a parameter of 0.

An Expose callback might look like this:

```
static void
exposeCB(w, client_data, call_data)
    Widget w;
    caddr_t client_data;
    GlxDrawCallbackStruct *call_data;
{
    GLXwinset(display, call_data->window);
    draw_scene();
    gflush();
}
```

The first parameter, *w*, is the widget. The second parameter, *client_data*, is passed in by the programmer when adding the callback. The third parameter, *call_data*, is provided by the widget itself. For the GlxDraw widget, this includes the reason for the callback, the window, and the window's width and height.

The first thing that the callback does is call **GLXwinset()**. Unlike X, where a window is passed to all drawing routines, the IRIS GL maintains the concept of a current window. **GLXwinset()** tells the IRIS GL to perform all subsequent operations in the specified window. Begin every callback by calling **GLXwinset()** to make sure you're dealing with the correct IRIS GL window. (The exception to this rule is when there is only one IRIS GL window in the application—in this case **GLXwinset()** is unnecessary.) The widget makes one call to **GLXwinset()** when it is created, and all subsequent IRIS GL operations go to that window.

After calling **GLXwinset()**, the callback redraws the image. In this example, that is accomplished by calling **draw_scene()**.

Finally, the call to **gflush()** flushes the queue if the application is running over the network. The call to **gflush()** can be in either the callback or in **draw_scene()**.

A resize callback is very similar:

```
static void
resizeCB(w, client_data, call_data)
    Widget w;
    caddr_t client_data;
    GlxDrawCallbackStruct *call_data;
{
    GLXwinset(display, call_data->window);
    viewport(0, (Screencoord) call_data->width-1,
        0, (Screencoord) call_data->height-1);
    gflush();
}
```

The main addition here is the call to **viewport()**. This tells the IRIS GL to resize the viewport to the same size as the window.

Using Overlays

The GlxDraw widget provides support for overlays and popups. Under X, overlay drawing is managed by the window system. In order to function in an environment where there might be X windows in the overlay planes, it is necessary for mixed-model programs to explicitly create a window in the overlay planes for overlay drawing.

To create an overlay, add the appropriate entries to the **GlxConfig()** resource, and set the *useOverlay* resource to TRUE. The widget creates the overlay window and automatically generates *overlayExpose* callbacks when necessary. To draw to the overlay at other times, get the window ID by querying the *overlayWindow* resource. Call **GLXwinset()** to select either the normal window or the overlay window as the drawing surface.

The last two steps in the outline for using GlxDraw, writing the IRIS GL code, and handling the input events are discussed in “Handling Input in a Mixed-Model Program” below.

Linking with the IRIS Widget Library

When using the IRIS GL widgets, link with *-lXirisw*. Here is an IRIS IM example:

```
ld -o progname prog.o -lXirisw -lXm_s -lXt_s -lX11_s -lPW -lsun -lmalloc
```

In the example, *progname* is the name of your program.

Handling Input in a Mixed-Model Program

To provide for smoother porting from system to system, as well as for easier integration of X and IRIS GL in a single application, always separate event handling loops from the rest of your program.

Input in Xt is event-driven. There are two ways of handling input with the GlxDraw widget. The first is to use the *input callback*, which provides a callback for keyboard and mouse events. The second is to use *actions* and *translations*, Xt-provided mechanisms that map keyboard input into user-provided routines.

Both the input callback and the translations have advantages. The input callback is usually somewhat simpler to write, especially the first time. Also with the input callback all input is handled by a single routine that can maintain private state.

On the other hand, the action and translation method can produce more modular programs, because translations have one function for each action. Also, with translations, the system does the keyboard parsing so you don't have to do it in the code. Finally, the use of translations allows the users to customize the bindings, for example, they can create a setup where they can type **q** instead of **<Esc>** to quit an application.

Using the Input Callback

By default, the input callback is called with every key press and release, with every mouse button press and release, and whenever the mouse is moved while a mouse button is pressed. You can change this by providing a different translation table, although the default setting should be suitable for many IRIS GL applications. The callback is passed an X event, which it should interpret, then perform the appropriate action. It is up to you to interpret the event—for example, to convert an X keycode into a key symbol—then decide what to do with it.

Using Actions and Translations

Actions and translations provide a mechanism for binding a key or mouse event to a function call. For example, you can set things up so that when the **<Esc>** key is pressed, exit is called; when mouse button 1 is pressed, a rotation occurs; and when **<f>** is pressed, the program zooms in. The following translations show how this might be done:

```
program*glwidget*translations:      #override \n\
    <Btn1Down>:        start_rotate()   \n\
    <Btn1Up>:          stop_rotate()    \n\
    <Btn1Motion>:      rotate()         \n\
    <Key>f:            zoom_in()        \n\
    <Key>b:            zoom_out()       \n\
    <KeyUp>osfCancel:   quit()
```

Although the syntax takes a little getting used to, the effect is clear. When button 1 is pressed, **start_rotate()** is called. When it is released, **stop_rotate()** is called. Moving the mouse with the button 1 causes the actual rotation. Similarly, the **<f>** and **<b>** keys cause zooming in and out.

The last entry is a little cryptic. It actually says that when the **<Esc>** key is pressed, **quit()** is called. However, the Open Software Foundation® has implemented *virtual bindings*, which allow the same programs to work on computers with different keyboards that may be missing various keys. If a key has a virtual binding, the virtual binding name must be specified in the translation. Thus, the example above specifies **osfCancel** rather than **`<Esc>`**. To use the above translation in a program that is not based on IRIS IM, replace **`<KeyUp>osfCancel`** with **`<KeyUp>Escape`**.

The translation is only half of what it takes to set up this binding. Although the translation table above has what looks like function names, they are actually *action* names. The program must create an *action table* to bind the action names to actual functions in the program. For more information on setting up actions and translations, see the recommended X Window System references.

Common Pitfalls

Regardless of whether you use the input callback or actions and translations, there is one simple pitfall to watch out for when parsing mouse events—namely that X and IRIS GL have different notions of the *y* direction. With X, positive *y* is down—with GL, positive *y* is up. It is easy to write an application where the program tries to track the mouse, only to find the object moving in the wrong direction vertically.

Another common problem that crops up is that, in programs using IRIS IM, it may appear that keyboard input is getting lost. This is caused by IRIS IM's traversal behavior. The keyboard input might actually be directed to another widget. There are two solutions to this. The easiest solution is to set the resource:

```
keyboardFocusPolicy: POINTER
```

for the application. This eliminates IRIS IM traversal, and always sets input focus to follow the pointer; however, doing so eliminates traversal for those users who prefer it and forces a nondefault model. A better solution is to set the resource:

```
traversalOn: TRUE
```

for the widget (not the application) and to call:

```
XmProcessTraversal(widget, XmTRAVERSE_CURRENT);
```

whenever mouse button 1 is pressed in the window. Turning *processTraversal* on causes the window to respond to traversal (which it normally does not), and calling **XmProcessTraversal()** actually traverses into the widget when needed.

Animation: Timeouts and Workprocs

In GL, animation is usually handled by a continuous loop, but because Xt is event-driven, this won't work in an Xt program. Instead, Xt provides two mechanisms, *timeouts* and *workprocs* (work procedures), that are useful in animation:

- A timeout is called every *n* milliseconds. It is useful for constant speed animation, as long as the processor can keep up.
- A workproc is called whenever the process has nothing else to do. It provides the highest speed animation that the processor can handle, but the speed varies depending on load.

One advantage of workprocs is that user actions such as menu postings have a higher priority than the workproc, so the menus will pop up immediately. Although the workproc has the lowest priority in the process, it still must compete with other processes. The example in "Mixed-Model Sample Programs Using Widgets and Xt" below shows how to use workprocs to produce animation.

Mixed-Model Sample Programs Using Widgets and Xt

This section presents two mixed-model programs that demonstrate the concepts used in this section. These examples are provided on line in */usr/people/4Dgifts/examples/GLX*. They are reprinted here for your convenience, but you should study the online versions because they can be updated after this guide is released.

Work Procedures and Popup Menus

Example 2-2 lists *wproc.c*, which shows how to achieve continuous animation through the Xt Intrinsics mechanism of workprocs. The program displays a continuously rotating wireframe cube. It also demonstrates how to create a popup menu for the IRIS GL widget. Click the right mouse button to see the popup menu.

Example 2-2 *wproc.c* Source Code

```
/** header *****************************************************************/
/*
//
// purpose:
//      mixed model motif program demonstrating
//      ... using workprocs for continuous animation
//      ... updating aspect of 3d view to keep "square’s square"
//      ... creating popup menu for the gl widget
//
// compiling:
//      cc -float -prototypes -DFUNCPROTO -O wproc.c -o wproc \
//      -s -lXirisw -lXm_s -lXt_s -lgl_s -lX11_s -lm -lsun -lPW
*/

/** notes ******************************************************************/
/** includes ***************************************************************/

#include <stdio.h>                      /* printf(), ... */
#include <Xm/Xm.h>                      /* for motif */
#include <Xm/Form.h>                    /* motif widget */
#include <Xm/Frame.h>                   /* motif widget */
#include <Xm/Label.h>                   /* motif widget */
#include <Xm/PushB.h>                   /* motif widget */
#include <Xm/RowColumn.h>               /* motif widget */
#include <Xm/Separator.h>               /* motif widget */
#include <X11/Xirisw/GlxMDraw.h>        /* gl widget */

/** defines ****************************************************************/

/* c environment */
#define global

/* colors */
#define RGB_BLACK       0x00000000
#define RGB_RED         0x000000FF
#define RGB_GREEN       0x0000FF00
#define RGB_BLUE        0x00FF0000

/** typedefs ***************************************************************/
/** prototypes *************************************************************/

extern void main(int argc, char *argv[], char *envp[]);

/* setup */
static void check_capabilities(void);
static void install_colormaps(Widget top_level, Widget glw);

/* callbacks (gl widget) */
static void gl_ginit_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void gl_expose_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void gl_resize_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void gl_input_cb(Widget w, XtPointer appdat, XtPointer sysdat);

/* callbacks (misc) */
static void quit_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void color_cb(Widget w, XtPointer appdat, XtPointer sysdat);

/* event handlers */
static void post_menu_eh(Widget w, Widget menu, XEvent *event);

/* work procedures */
static Boolean anim_wp(XtPointer appdat);

/* drawing */
static void draw_frame(char *ops);
static void model_cube_wire(void);

/** variables **************************************************************/

/* fallback resources */
static char *fallback_resources[] = {
    "Wproc*Red*foreground: red",
    "Wproc*Green*foreground: green4",
    "Wproc*Blue*foreground: blue",
    "Wproc*info_label*labelString: "
        "[ Use the Right Mouse Button to pop up color menu ]",
    NULL,
};
```

```
/*
// mixed-model configuration:
*/
static GLXconfig glx_config[] = {
    {GLX_NORMAL, GLX_DOUBLE, TRUE},
    {GLX_NORMAL, GLX_RGB, TRUE},
    { 0, 0, 0 },
};
static unsigned long cube_color = RGB_GREEN;

/** functions ***************************************************************/
/*
// main - program entry point.
*/
global void main(
    int argc,                   /* argument count */
    char *argv[],               /* argument vector */
    char *envp[]                /* environment pointer */
)
{
    XtAppContext app_context;   /* application context */
    Display *dsp;               /* display ref */
    Widget app_shell;           /* first widget */
    Widget form;                /* surrounds app */
    Widget rowcol;              /* manages input buttons */
    Widget button;              /* utility button */
    Widget label;               /* utility label */
    Widget separator;           /* utility separator */
    Widget frame;               /* to surround gl widget */
    Widget glw;                 /* can do gl rendering in this guy */
    Widget menu;                /* simple popup for gl widget */
    XtWorkProcId anim_wpid;     /* animation work proc */
    Arg args[15];               /* for name/value pairs */
    int n;                      /* reusable indices */

    /* make sure we can we do it */
    check_capabilities();

    /* initialize toolkit, creating application shell */
    n = 0;
    XtSetArg(args[n], XmNtitle, "Work Proc"); n++;
    app_shell = XtAppInitialize(
        &app_context, "Wproc", NULL, 0, &argc, argv,
        fallback_resources, args, n
                                  );
```

```
/* create container for app */
n = 0;
form = XmCreateForm(app_shell, "form", args, n);
XtManageChild(form);

/* create the command area */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
rowcol = XmCreateRowColumn(form, "rowcol", args, n);
XtManageChild(rowcol);

/* create the command area buttons */
n = 0;
button = XmCreatePushButton(rowcol, "Quit", args, n);
XtAddCallback(button, XmNactivateCallback, quit_cb, NULL);
XtManageChild(button);

/* create separator between command area and output area */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(args[n], XmNleftWidget, rowcol); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
separator = XmCreateSeparator(form, "separator", args, n);
XtManageChild(separator);

/* create the output area */
/* create the informational label */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(args[n], XmNleftWidget, separator); n++;
XtSetArg(args[n], XmNleftOffset, 5); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNrightOffset, 5); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopOffset, 5); n++;
label = XmCreateLabel(form, "info_label", args, n);
XtManageChild(label);
```

```
    /* create the frame */
    n = 0;
    XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
    XtSetArg(args[n], XmNleftWidget, separator); n++;
    XtSetArg(args[n], XmNleftOffset, 5); n++;
    XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNrightOffset, 5); n++;
    XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNbottomOffset, 5); n++;
    XtSetArg(args[n], XmNtopAttachment, XmATTACH_WIDGET); n++;
    XtSetArg(args[n], XmNtopWidget, label); n++;
    XtSetArg(args[n], XmNtopOffset, 5); n++;
    XtSetArg(args[n], XmNshadowThickness, 6); n++;
    frame = XmCreateFrame(form, "frame", args, n);
    XtManageChild(frame);

    /* create the gl widget */
    n = 0;
    XtSetArg(args[n], GlxNglxConfig, glx_config); n++;
    XtSetArg(args[n], XmNborderWidth, 0); n++;
    XtSetArg(args[n], XmNwidth, 400); n++;
    XtSetArg(args[n], XmNheight, 400); n++;
    glw = GlxCreateMDraw(frame, "glw", args, n);
    XtManageChild(glw);
    XtAddCallback(glw, GlxNginitCallback, gl_ginit_cb, 0);
    XtAddCallback(glw, GlxNexposeCallback, gl_expose_cb, 0);
    XtAddCallback(glw, GlxNresizeCallback, gl_resize_cb, 0);
    XtAddCallback(glw, GlxNinputCallback, gl_input_cb, 0);

    /* create a popup menu */
    n = 0;
    menu = XmCreatePopupMenu(form, "menu", args, n);
    XtAddEventHandler(
        form, ButtonPressMask, FALSE, (XtEventHandler) post_menu_eh,
        (XtPointer) menu
    );

    /* menu title is the name of the program */
    n = 0;
    label = XmCreateLabel(menu, "Color", args, n);
    XtManageChild(label);
    separator = XmCreateSeparator(menu, "separator", args, n);
    XtManageChild(separator);
    separator = XmCreateSeparator(menu, "separator", args, n);
    XtManageChild(separator);
    /* add some buttons to change color */
    n = 0;
    button = XmCreatePushButton(menu, "Red", args, n);
    XtAddCallback(button, XmNactivateCallback, color_cb, (XtPointer) RGB_RED);
    XtManageChild(button);
    button = XmCreatePushButton(menu, "Green", args, n);
    XtAddCallback(button, XmNactivateCallback, color_cb, (XtPointer) RGB_GREEN);
    XtManageChild(button);
    button = XmCreatePushButton(menu, "Blue", args, n);
    XtAddCallback(button, XmNactivateCallback, color_cb, (XtPointer) RGB_BLUE);
    XtManageChild(button);

    /* setup work procedure */
    anim_wpid = XtAppAddWorkProc(app_context, anim_wp, (XtPointer) glw);

    /* realize the app, creating the actual x windows */
    XtRealizeWidget(app_shell);
    install_colormaps(app_shell, glw);

    /* enter the event loop */
    XtAppMainLoop(app_context);
}

/*- support: setup --------------------------------------------------------*/
/*
// check_capabilities - find out if the machine can do what we need.
*/
static void check_capabilities(void)
{
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
        fprintf(stderr, "Double buffered RGB mode not available.\n");
        exit(1);
    }
}
```

```
/*
// install_colormaps - let the window manager know about our colormaps.
//
// This has been generalized to handle any windows a gl widget might have.
// It may not necessarily being using any of them.
*/
static void install_colormaps(Widget top_level, Widget glw)
{
    Window overlay_win, popup_win, underlay_win;
    Window window[5];
                                   int i;

    XtVaGetValues(
        glw,
        GlxNoverlayWindow, &overlay_win,
        GlxNpopupWindow, &popup_win,
        GlxNunderlayWindow, &underlay_win,
        NULL
    );
    i = 0;
    if (overlay_win)
        window[i++] = overlay_win;
    if (popup_win)
        window[i++] = popup_win;
    if (underlay_win)
        window[i++] = underlay_win;
    window[i++] = XtWindow(glw);
    window[i++] = XtWindow(top_level);
    XSetWMColormapWindows(XtDisplay(top_level), XtWindow(top_level), window, i);
}
```

```
/*- support: callbacks (gl widget) -----------------------------------------*/
/*
// gl_ginit_cb - perform any necessary graphics initialization.
*/
static void gl_ginit_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;

    GLXwinset(XtDisplay(w), XtWindow(w));
    mmode(MVIEWING);
    perspective(300, glx->width/(float)glx->height, 1.0, 50.0);
    polarview(10.0, 0, 0, 0);
    frontbuffer(TRUE);
    cpack(RGB_BLACK);
    clear();
    frontbuffer(FALSE);
    gflush();
}

/*
// gl_expose_cb - handle expose events for the gl widget.
*/
static void gl_expose_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;

    GLXwinset(XtDisplay(w), XtWindow(w));
    draw_frame("cds");
}


/*
// gl_resize_cb - handle resize events for the gl widget.
*/
static void gl_resize_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;

    GLXwinset(XtDisplay(w), XtWindow(w));
    viewport(0, glx->width-1, 0, glx->height-1);
    perspective(300, glx->width/(float)glx->height, 1.0, 50.0);
}
```

```
/*
// gl_input_cb - handle input from a gl window.
*/
static void gl_input_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;

    GLXwinset(XtDisplay(w), XtWindow(w));
}

/*- support: callbacks (misc) ----------------------------------------------*/
/*
// quit_cb - exit application.
*/
static void quit_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    exit(0);
}

/*
// color_cb - change cube color.
*/
static void color_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    cube_color = (unsigned long) appdat;
}

/*- support: event handlers ------------------------------------------------*/
/*
// post_menu_eh - popup menu to get user’s color selection.
*/
static void post_menu_eh(Widget w, Widget menu, XEvent *event)
{
    int button;

    /* make sure it’s the correct button being pressed */
    XtVaGetValues(menu, XmNwhichButton, &button, NULL);
    if (event->xbutton.button == button) {
        XmMenuPosition(menu, (XButtonPressedEvent *) event);
        XtManageChild(menu);
    }
}
```

```
/*- support: work procedures ---------------------------------------------*/
/*
// anim_wp - do another frame of animation.
*/
static Boolean anim_wp(XtPointer appdat)
{
    Widget glw = (Widget) appdat;

    GLXwinset(XtDisplay(glw), XtWindow(glw));
    draw_frame("cdsu");
    return (False);
}

/*- support: drawing -------------------------------------------------------*/
/*
// draw_frame - localize steps to drawing a frame.
*/
static void draw_frame(char *ops)
{
    static Angle rx = 0;
    static Angle ry = 0;
    static Angle rz = 0;
    for (; *ops != ’\0’; ops++) {
        switch (ops[0]) {
        case ’c’:               /* clear */
            cpack(RGB_BLACK);
            clear();
            break;
        case ’d’:               /* draw */
            cpack(cube_color);
            pushmatrix();
                rotate(rz, ’z’);
                rotate(ry, ’y’);
                rotate(rx, ’x’);
                model_cube_wire();
            popmatrix();
            break;
        case ’s’:               /* swap */
            swapbuffers();
            gflush();
            break;
        case ’u’:               /* update */
            /* next angle */
            rx = (rx + 10) % 3600;
            ry = (ry + 10) % 3600;
            rz = (rz + 10) % 3600;
            break;
        default:
            break;
        }
    }
}

/*- support: modelling primitives -----------------------------------------*/
/*
// model_cube_wire - cube primitive (3D, wireframe)
*/
static void model_cube_wire(void)
{
    static long v[8][3] = {
        {-1, -1, -1},
        {-1, -1,  1},
        {-1,  1,  1},
        {-1,  1, -1},
        { 1, -1, -1},
        { 1, -1,  1},
        { 1,  1,  1},
        { 1,  1, -1},
    };
    static int path[16] = {
        0, 1, 2, 3,
        0, 4, 5, 6,
        7, 4, 5, 1,
        2, 6, 7, 3
    };
    int i;

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

/** eof ********************************************************************/
```

Mouse Input and Colormaps

Example 2-3 lists *cmapov.c,* which installs a colormap and uses mouse input.

Example 2-3 *cmapov.c* Source Code

```
/** header *****************************************************************/
/*
//
// purpose:
//      mixed model program demonstrating
//      ...using custom colormaps for the normal and overlay buffers.
//      ...moving things with the mouse.
//
// compiling:
//      cc -float -prototypes -DFUNCPROTO -O cmapov.c -o cmapov
//      -s -lXirisw -lXm_s -lXt_s -lgl_s -lX11_s -lm -lsun -lPW
//
// operating:
//      Use the left mouse button to move the red, green, and blue blocks.
//      Verify that they "pass under" the yellow, magenta, and cyan blocks
//      which are in the overlay planes.
//
*/

/** includes ***************************************************************/

#include <stdio.h>                      /* standard */
#include <Xm/Xm.h>                      /* for motif */
#include <Xm/Form.h>                    /* motif widget */
#include <Xm/Frame.h>                   /* motif widget */
#include <Xm/PushB.h>                   /* motif widget */
#include <Xm/RowColumn.h>               /* motif widget */
#include <Xm/Separator.h>               /* motif widget */
#include <X11/Xirisw/GlxMDraw.h>        /* gl widget */

/** defines ****************************************************************/

/* c environment */
#define global

/** typedefs ***************************************************************/
/** prototypes **************************************************************/

extern void main(int argc, char *argv[], char *envp[]);

/* setup */

static void check_capabilities(void);
static void  install_colormaps(Widget top_level, Widget glw);
static void  normal_cmap_init(Widget glw);
static Pixel normal_cmap_set(Widget glw, int index, short r, short g, short b);
static void  overlay_cmap_init(Widget glw);
static Pixel overlay_cmap_set(Widget glw, int index, short r, short g, short b);

/* mixed model support */
static void gl_ginit_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void gl_expose_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void gl_resize_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void gl_input_cb(Widget w, XtPointer appdat, XtPointer sysdat);
static void gl_overlay_expose_cb(Widget w, XtPointer appdat, XtPointer sysdat);

/* callbacks (misc) */
static void quit_cb(Widget w, XtPointer appdat, XtPointer sysdat);

/* drawing */
static void draw_normal_frame(void);
static void draw_overlay_frame(void);
static void draw_boxes(int c1, int c2, int c3);

/** variables ***************************************************************/
/* mixed-model configuration */
static GLXconfig glx_config[] = {
    {GLX_NORMAL, GLX_DOUBLE, TRUE},
    {GLX_OVERLAY, GLX_BUFSIZE, 2},
    { 0, 0, 0 },
};
```

```
/* information which allows us to use the overlay or popup buffer */
static struct {
    char *use;
    char *expose_cb;
    char *window;
    char *visual;
    char *colormap;
} *over_res, over_res_map[2] = {
    /* describe needed overlay resources */
    {   GlxNuseOverlay, GlxNoverlayExposeCallback, GlxNoverlayWindow,
        GlxNoverlayVisual, GlxNoverlayColormap
    },
    /* describe analogous popup resources for when overlays aren't there */
    {   GlxNusePopup, GlxNpopupExposeCallback, GlxNpopupWindow,
        GlxNpopupVisual, GlxNpopupColormap
    },
};

/* normal buffer colors */
static Pixel n_grey, n_red, n_green, n_blue;

/* overlay buffer colors */
static Pixel o_trans, o_yellow, o_magenta, o_cyan;

/* gl window info */
static struct {
    Dimension width;            /* in pixels */
    Dimension height;           /* in pixels */
    float pt[3];                /* world position of moving object */
} glwin = {400, 400, {15.0, 20.0, 0.0}};

/** functions *************************************************************/

/*
// main - program entry point.
*/
global void main(
    int argc,                   /* argument count */
    char *argv[],               /* argument vector */
    char *envp[]                /* environment pointer */
)
```

```
{
    XtAppContext app_context;   /* application context */
    Widget app_shell;           /* first widget */
    Widget form;                /* surrounds app */
    Widget rowcol;              /* manages input buttons */
    Widget button;              /* quit button */
    Widget separator;           /* between input and output */
    Widget frame;               /* to surround gl widget */
    Widget glw;                 /* the gl widget inside window */
    Arg args[15];               /* for name/value pairs */
    int n;                      /* for reusable indices */

    /* initialize toolkit, creating application shell */
    n = 0;
    XtSetArg(args[n], XmNtitle, "CMode Overlay"); n++;
    app_shell = XtAppInitialize(
        &app_context, "Cmapov", NULL, 0, &argc, argv, NULL,
        args, n
    );

    /* create container for app */
    n = 0;
    form = XmCreateForm(app_shell, "form", args, n);
    XtManageChild(form);

    /* create the command area */
    n = 0;
    XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
    rowcol = XmCreateRowColumn(form, "rowcol", args, n);
    XtManageChild(rowcol);

    /* create the command area buttons */
    n = 0;
    button = XmCreatePushButton(rowcol, "Quit", args, n);
    XtAddCallback(button, XmNactivateCallback, quit_cb, NULL);
    XtManageChild(button);
```

```
/* create separator between command area and output area */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(args[n], XmNleftWidget, rowcol); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNorientation, XmVERTICAL); n++;
separator = XmCreateSeparator(form, "separator", args, n);
XtManageChild(separator);
/* create the output area */
/* create the frame */
n = 0;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_WIDGET); n++;
XtSetArg(args[n], XmNleftWidget, separator); n++;
XtSetArg(args[n], XmNleftOffset, 5); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNrightOffset, 5); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNbottomOffset, 5); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNtopOffset, 5); n++;
XtSetArg(args[n], XmNshadowThickness, 6); n++;
frame = XmCreateFrame(form, "frame", args, n);
XtManageChild(frame);
/* create the gl widget */
n = 0;
XtSetArg(args[n], GlxNglxConfig, glx_config); n++;
XtSetArg(args[n], over_res->use, True); n++;
XtSetArg(args[n], XmNborderWidth, 0); n++;
XtSetArg(args[n], XmNwidth, glwin.width); n++;
XtSetArg(args[n], XmNheight, glwin.height); n++;
glw = GlxCreateMDraw(frame, "glw", args, n);
XtManageChild(glw);
XtAddCallback(glw, GlxNginitCallback, gl_ginit_cb, 0);
XtAddCallback(glw, GlxNexposeCallback, gl_expose_cb, 0);
XtAddCallback(glw, GlxNresizeCallback, gl_resize_cb, 0);
XtAddCallback(glw, GlxNinputCallback, gl_input_cb, 0);
XtAddCallback(glw, over_res->expose_cb, gl_overlay_expose_cb, 0);
```

```
    /* setup custom normal colormap */
    normal_cmap_init(glw);
    n_grey  = normal_cmap_set(glw, 0, 125, 125, 125);
    n_red   = normal_cmap_set(glw, 1, 255,   0,   0);
    n_green = normal_cmap_set(glw, 2,   0, 255,   0);
    n_blue  = normal_cmap_set(glw, 3,   0,   0, 255);

    /* setup custom overlay colormap */
    overlay_cmap_init(glw);
    o_trans   = 0;      /* transparent is always zero */
    o_yellow  = overlay_cmap_set(glw, 0, 255, 255,   0);
    o_magenta = overlay_cmap_set(glw, 1, 255,   0, 255);
    o_cyan    = overlay_cmap_set(glw, 2,   0, 255, 255);

    /* realize the app, creating the actual x windows */
    XtRealizeWidget(app_shell);

    /* setup for colormap installation */
    install_colormaps(app_shell, glw);

    /* enter the event loop */
    XtAppMainLoop(app_context);
}

/*- support: setup ---------------------------------------------------------*/

/*
// check_capabilities - find out if the machine can do what we need.
*/
static void check_capabilities(void)
{
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
        fprintf(stderr, "Double buffered RGB mode not available.\n");
        exit(1);
    }
    /* use popup planes if there is not enough overlay planes */
    over_res = &over_res_map[0];
    if (getgdesc(GD_BITS_OVER_SNG_CMODE) < 2) {
        glx_config[1].buffer = GLX_POPUP;
        over_res = &over_res_map[1];
    }
    printf(
        "\nUsing the %s planes\n", over_res==over_res_map? "OVERLAY" : "POPUP"
    );
}

/*
// install_colormaps - let the window manager know about our colormaps.
//
// This has been generalized to handle any windows a gl widget might have.
// It may not necessarily being using any of them.
*/
static void install_colormaps(Widget top_level, Widget glw)
{
    Window overlay_win, popup_win, underlay_win;
    Window window[5];
    int i;

    XtVaGetValues(
        glw,
        GlxNoverlayWindow, &overlay_win,
        GlxNpopupWindow, &popup_win,
        GlxNunderlayWindow, &underlay_win,
        NULL
    );
    i = 0;
    if (overlay_win)
        window[i++] = overlay_win;
    if (popup_win)
        window[i++] = popup_win;
    if (underlay_win)
        window[i++] = underlay_win;
    window[i++] = XtWindow(glw);
    window[i++] = XtWindow(top_level);
    XSetWMColormapWindows(XtDisplay(top_level), XtWindow(top_level), window, i);
}
```

```
/*- support: custom normal colormap ---------------------------------------*/
/*
// normal_cmap_init - create a new normal colormap for the gl widget.
//
// The gl widget must already be created prior to calling this function,
// however the gl widget does not need to be realized for it to work.  This
// is because the window it uses in creating the colormap is the root window
// on the same screen.
*/
static void normal_cmap_init(Widget glw)
{
    Display *display;
    Window window;
    XVisualInfo *visinfo;
    Colormap pmap;
    Colormap cmap;
    XColor *color;
    int ncolors;
    int i;

    /* get display; any window on the same screen; and the visual */
    display = XtDisplay(glw);
    window = RootWindowOfScreen(XtScreen(glw));
    XtVaGetValues(glw, XmNvisual, &visinfo, NULL);

    /* create new normal colormap, allocating all entries */
    cmap = XCreateColormap(display, window, visinfo->visual, AllocAll);

    /* set new normal colormap for the gl widget */
    XtVaSetValues(glw, XmNcolormap, cmap, NULL);

    /*
    // duplicate the parent’s default colors for the lower colormap entries
    // (max 256) to avoid colormap flashing on machines with only one h/w
    // colormap.
    */
    XtVaGetValues(XtParent(glw), XmNcolormap, &pmap, NULL);
    ncolors = visinfo->colormap_size;
    printf("\nnormal colors = %d\n", ncolors);
    if (ncolors > 256)
        ncolors = 256;
    color = (XColor *) XtMalloc(ncolors*sizeof(XColor));
    for (i=0; i<ncolors; i++)
        color[i].pixel = i;
    XQueryColors(display, pmap, color, ncolors);
    XStoreColors(display, cmap, color, ncolors);
    XtFree((char *)color);
}

/*
// normal_cmap_set - map a color for the normal buffer.
//
// This uses a simple scheme of mapping the colors backwards from the highest
// colormap index.
*/
static Pixel normal_cmap_set(Widget glw, int index, short r, short g, short b)
{
    XVisualInfo *visinfo;
    Colormap cmap;
    XColor color;
    int n_last;
    XtVaGetValues(glw, XmNvisual, &visinfo, XmNcolormap, &cmap, NULL);
    n_last = visinfo->colormap_size-1;
    color.pixel = n_last - index; /* work backwards from the last position */
    color.flags = DoRed | DoGreen | DoBlue;
    color.red   = r << 8;
    color.green = g << 8;
    color.blue  = b << 8;
    XStoreColor(XtDisplay(glw), cmap, &color);
    return (color.pixel);
}
```

```
/*- support: custom overlay colormap ---------------------------------------*/
/*
// overlay_cmap_init - create a new overlay colormap for the gl widget.
//
// The gl widget must already be created prior to calling this function,
// however the gl widget does not need to be realized for it to work.  This
// is because the window it uses in creating the colormap is the root window
// on the same screen.
*/
static void overlay_cmap_init(Widget glw)
{
    Display *display;
    Window window;
    XVisualInfo *visinfo;
    Colormap cmap;
    XColor color;
    int ncolors;
    Pixel *pixel;
    unsigned long plane_mask[1];
    int result;

    /* get display; any window on the same screen; and the visual */
    display = XtDisplay(glw);
    window = RootWindowOfScreen(XtScreen(glw));
    XtVaGetValues(glw, over_res->visual, &visinfo, NULL);

    /*
     * create new overlay colormap, allocating no entries.
     * (AllocAll would fail here because index 0 is reserved for transparency)
     */
    cmap = XCreateColormap(display, window, visinfo->visual, AllocNone);

    /* set new overlay colormap for the gl widget */
    XtVaSetValues(glw, over_res->colormap, cmap, NULL);

    /* allocate every color except transparency as read/write */
    ncolors = visinfo->colormap_size;   /* including transparent color */
    printf("\noverlay colors = %d\n", ncolors);
    pixel = (Pixel *) XtMalloc(ncolors*sizeof(Pixel));  /* stub array */
    result = XAllocColorCells(
        display, cmap, True, plane_mask, 0,
        &pixel[1], ncolors-1    /* one less due to transparency */
    );
    XtFree((char *) pixel);
    /* check for booboo */
    if (result == 0)
        fprintf(stderr, "XAllocColorCells failed for overlay buffer.\n");
}


/*
// overlay_cmap_set - map a color for the overlay buffer.
//
// This uses a simple scheme of mapping the colors backwards from the highest
// colormap index.
*/
static Pixel overlay_cmap_set(Widget glw, int index, short r, short g, short b)
{
    XVisualInfo *visinfo;
    Colormap cmap;
    XColor color;
    int n_last;

    XtVaGetValues(
        glw, over_res->visual, &visinfo, over_res->colormap, &cmap, NULL
    );
    n_last = visinfo->colormap_size-1;
    color.pixel = n_last - index; /* work backwards from the last position */
    color.flags = DoRed | DoGreen | DoBlue;
    color.red   = r << 8;
    color.green = g << 8;
    color.blue  = b << 8;
    XStoreColor(XtDisplay(glw), cmap, &color);
    return (color.pixel);
}


/*- support: callbacks (gl widget) -----------------------------------------*/
/*
// gl_ginit_cb - perform any necessary graphics initialization.
*/
static void gl_ginit_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;
    GLXwinset(XtDisplay(w), XtWindow(w));
    mmode(MVIEWING);
    ortho2(-0.5, 100.5, -0.5, 100.5);
    gflush();
}

/*
// gl_expose_cb - handle expose events for the gl widget.
*/
static void gl_expose_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;

    GLXwinset(XtDisplay(w), XtWindow(w));
    draw_normal_frame();
}


/*
// gl_resize_cb - handle resize events for the gl widget.
*/
static void gl_resize_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;
    Window overlay_window;

    /* squirrel away size */
    glwin.width = glx->width;
    glwin.height = glx->height;

    /* setup normal buffer viewport */
    GLXwinset(XtDisplay(w), XtWindow(w));
    viewport(0, glx->width-1, 0, glx->height-1);

    /* setup overlay buffer viewport */
    XtVaGetValues(w, over_res->window, &overlay_window, NULL);
    GLXwinset(XtDisplay(w), overlay_window);
    viewport(0, glx->width-1, 0, glx->height-1);
}
```

```
/*
// gl_input_cb - handle input for the gl window.
*/
static void gl_input_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    static Boolean active = False;      /* currently moving? */
    static float dx, dy;                /* offset from current position */
    /**/
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;
    XEvent *event = glx->event; /* what occured */
    int msx, msy;               /* gl window mouse position */
    float mwx, mwy;             /* gl world  mouse position */

    GLXwinset(XtDisplay(w), XtWindow(w));
    /* map to gl window coords */
    msx = event->xbutton.x;                     /* same x */
    msy = (glwin.height-1) - event->xbutton.y;  /* flip y */
    /* map to gl world coords */
    mwx = 0.0 + ((msx - 0) / (float)glwin.width ) * 100.0;
    mwy = 0.0 + ((msy - 0) / (float)glwin.height) * 100.0;
    /* process event */
    switch (event->type) {
    case ButtonPress:
        if (event->xbutton.button == Button1) {
            /* compute delta from current position */
            dx = mwx - glwin.pt[0];
            dy = mwy - glwin.pt[1];
            active = True;
        }
        break;
    case MotionNotify:
        if (active) {
            /* compute new position and draw */
            glwin.pt[0] = mwx - dx;
            glwin.pt[1] = mwy - dy;
            draw_normal_frame();
        }
        break;
    case ButtonRelease:
        if (event->xbutton.button == Button1) {
            /* we’re done */
            active = False;
        }
        break;
    }
}

/*
// gl_overlay_expose_cb - handle overlay expose events for the gl widget.
*/
static void gl_overlay_expose_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    GlxDrawCallbackStruct *glx = (GlxDrawCallbackStruct *) sysdat;

    GLXwinset(XtDisplay(w), glx->window);
    draw_overlay_frame();
}

/*- support: callbacks (misc) ---------------------------------------------*/
/*
// quit_cb - exit application.
*/
static void quit_cb(Widget w, XtPointer appdat, XtPointer sysdat)
{
    exit(0);
}

/*- support: drawing -------------------------------------------------------*/
/*
// draw_normal_frame - render objects in the normal buffer and swap.
*/
static void draw_normal_frame(void)
{
    color(n_grey);
    clear();
    pushmatrix();
        translate(glwin.pt[0], glwin.pt[1], glwin.pt[2]);
        draw_boxes(n_red, n_green, n_blue);
    popmatrix();
    swapbuffers();
    gflush();
}
```

```
/*
// draw_overlay_frame - render objects in the overlay buffer.
*/
static void draw_overlay_frame(void)
{
    color(o_trans);
    clear();
    pushmatrix();
        translate(15.0, 60.0, 0.0);
        draw_boxes(o_yellow, o_cyan, o_magenta);
    popmatrix();
    gflush();
}


/*
// draw_boxes - draw three boxes in three different colors.
*/
static void draw_boxes(int c1, int c2, int c3)
{
    static float vert[][2] = {  /* a box */
        { 0.0,  0.0},
        {20.0,  0.0},
        {20.0, 20.0},
        { 0.0, 20.0},
    };

    pushmatrix();
    color(c1);
    bgnpolygon();
        v2f(vert[0]); v2f(vert[1]); v2f(vert[2]); v2f(vert[3]);
    endpolygon();
    translate(25.0, 0.0, 0.0);
    color(c2);
    bgnpolygon();
        v2f(vert[0]); v2f(vert[1]); v2f(vert[2]); v2f(vert[3]);
    endpolygon();
    translate(25.0, 0.0, 0.0);
    color(c3);
    bgnpolygon();
        v2f(vert[0]); v2f(vert[1]); v2f(vert[2]); v2f(vert[3]);
    endpolygon();
    popmatrix();
}
/** eof ******************************************************************/
```

Using Xlib to Write a GLX Program

This section explains how to use Xlib and the four special GLX routines to create a mixed-model program.

Configuring an X Window for IRIS GL Rendering

The IRIS GL cannot draw into any ordinary X window. A window with the appropriate X visual for IRIS GL rendering must be created. There is no method for determining from the X interface which of the many available visuals to choose; therefore, you must use the **GLXgetconfig()** routine to obtain the proper visual.

Using the GLX Mixed-Model Routines

Silicon Graphics provides four routines designed specifically for mixed-model programming. This section provides a brief overview of these routines. For more detailed information, refer to the man pages.

Window Configuration: GLXgetconfig()

Use **GLXgetconfig()** to configure an X window for IRIS GL rendering. **GLXgetconfig()** takes the display, screen, and configuration information and returns the data needed to create and render IRIS GL into an X window. The return value is a complete description of the actual configuration available. This is useful to check what configuration was available, but it is also needed as an argument to **GLXlink()**. It can be freed with **free**(3) when it is no longer needed. **GLXgetconfig()** performs a role in mixed-model programs similar to the role of **gconfig()** in pure IRIS GL programs.

See "Declaring the GLXconfig Resource" on page 35 and the *GLXgetconfig*(3G) man page for a description of the GLXconfig structure.

Rendering IRIS GL in a Window: GLXlink()

Once you create a window, use **GLXlink()** to communicate to the IRIS GL that you intend to render into it. See the **GLXlink**(3G) man page for more detailed information about using **GLXlink()**.

Drawing: GLXwinset()

Once you select a window for IRIS GL rendering using **GLXlink()**, you can direct IRIS GL drawing commands to the window with the call **GLXwinset()**. This call indicates that all subsequent IRIS GL drawing commands will happen in the window passed to **GLXwinset()**. You can switch quickly between several windows by calling **GLXwinset()** for each window. **GLXwinset()** also selects the appropriate drawmode for the window. See the **GLXwinset**(3G) man page for more detailed information about using **GLXwinset()**.

Cleanup: GLXunlink()

A small amount of memory is allocated when a window is selected for IRIS GL rendering. For completeness, the routine **GLXunlink()** is provided to allow the IRIS GL to clean up the resources connected with IRIS GL rendering for a window. If you destroy a window with **XDestroyWindow()**, you should call **GLXunlink()** afterward. Typically, you call **GLXunlink()** only if a program will not do any IRIS GL drawing for a long time. See the *GLXunlink*(3G) man page for more detailed information about using **GLXunlink()**.

Mixed-Model Example Program Using Xlib and IRIS GL

Example 2-4 provides an example of a mixed-model program that uses double buffered RGB and overlays, based on Xlib. This example is on line in */usr/people/4Dgifts/examples/GLX*. Check the on-line version for updates before using this code.

Example 2-4 An Example Using Xlib and IRIS GL

```
/*
 *   mixexamp.c:
 *   This program is an example of a mixed model program that uses double
 *   buffered RGB and overlays (or popups if no overlays are present),
 *   based on Xlib.
 *   To compile:    cc -o mixexamp mixexamp.c -lgl_s -lX11_s
  */

/* Include X headers files first */
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <gl/glws.h>

/* X Display */
Display* D;

/* To use for X screen */
int S;

struct {
    short       vertex[2];
    short       color[3];
} stamp[4] = {
        { { 0, 0 }, { 255,   0,   0 } },
        { { 0, 1 }, {   0, 255,   0 } },
        { { 1, 1 }, {   0,   0, 255 } },
        { { 1, 0 }, { 255, 255,   0 } }
};

/* Declare the data structure for the IRIS GL rendering configuration needed */
GLXconfig rgb_ov[] = {
        { GLX_NORMAL,   GLX_RGB,        True} ,
        { GLX_OVERLAY,  GLX_BUFSIZE,    2} ,
        { 0,            0,              0}
};
/* use this one if we find we're on a machine that does not have overlays */
GLXconfig rgb_pup[] = {
        { GLX_NORMAL,   GLX_RGB,        True} ,
        { GLX_POPUP,    GLX_BUFSIZE,    2} ,
        { 0,            0,              0}
};
```

```
unsigned long
extract_value(int buffer, int mode, GLXconfig *conf)
{
    int i;
    for (i = 0; conf[i].buffer; i++)
        if (conf[i].buffer == buffer && conf[i].mode == mode)
            return conf[i].arg;
    return 0;
}

/* Extract X visual information */
XVisualInfo*
extract_visual(int buffer, GLXconfig *conf)
{
    XVisualInfo template, *v;
    int n;

    template.screen = S;
    template.visualid = extract_value(buffer, GLX_VISUAL, conf);
    return XGetVisualInfo (D, VisualScreenMask|VisualIDMask, &template, &n);
}

/* Fill the configuration structure with the appropriately */
/* created window */
void
set_window(int buffer, Window W, GLXconfig *conf)
{
    int i;

    for (i = 0; conf[i].buffer; i++)
        if (conf[i].buffer == buffer && conf[i].mode == GLX_WINDOW)
            conf[i].arg = W;
}


main(argc, argv)
int argc;
char *argv[];
{
    GLXconfig                *conf;
    XVisualInfo*             v;
    XSetWindowAttributes     attr;
    Window                   W, ovW, wins[2];
    int                      i;
    int                      OverlayPlanes;
    /* Follow the DISPLAY environment variable */
    D = XOpenDisplay(0);
    S = DefaultScreen(D);

    if (getgdesc(GD_BITS_OVER_SNG_CMODE) < 2) {/* test for overlay planes */
        if ((conf = GLXgetconfig(D, S, rgb_pup)) == 0) {
            printf("getconfig failed\n");
            exit(1);
        }
        OverlayPlanes = FALSE;
    } else {
        if ((conf = GLXgetconfig(D, S, rgb_ov)) == 0) {
            printf("getconfig failed\n");
            exit(1);
        }
        OverlayPlanes = TRUE;
    }
        /* Turn off verbose Xlib error messages */
    XSetErrorHandler(0);

        /* Create main plane window */
    v = extract_visual(GLX_NORMAL, conf);
    attr.colormap = extract_value(GLX_NORMAL, GLX_COLORMAP, conf);
    attr.border_pixel = 0;
    W = XCreateWindow(D, RootWindow(D, S), 0, 0, 400, 400, 0,
                      v->depth, InputOutput, v->visual,
                      CWBorderPixel|CWColormap, &attr);
    XStoreName(D, W, "DBL-Bufr'd, Overlay/Popup Mix Model Window");
    set_window(GLX_NORMAL, W, conf);

        /* Create overlay window, or popup if no overlay on this machine */
    if (OverlayPlanes) {
        v = extract_visual(GLX_OVERLAY, conf);
        attr.colormap = extract_value(GLX_OVERLAY, GLX_COLORMAP, conf);
        ovW = XCreateWindow(D, W, 0, 0, 400, 400, 0,
                            v->depth, InputOutput, v->visual,
                            CWBorderPixel|CWColormap, &attr);
        set_window(GLX_OVERLAY, ovW, conf);
    } else {
        v = extract_visual(GLX_POPUP, conf);
        attr.colormap = extract_value(GLX_POPUP, GLX_COLORMAP, conf);
        ovW = XCreateWindow(D, W, 0, 0, 400, 400, 0,
                            v->depth, InputOutput, v->visual,
                            CWBorderPixel|CWColormap, &attr);
        set_window(GLX_POPUP, ovW, conf);
    }
        /* Bind the IRIS GL to the created windows */
    if (GLXlink(D, conf) < 0) {
        printf(“Bind failed\n”);
        exit(1);
    }
   wins[0] = W;
    wins[1] = ovW;
    XSetWMColormapWindows(D, W, wins, 2);
    XSelectInput(D, W, KeyPressMask|ExposureMask);
    XMapWindow(D, W);
    XSelectInput(D, ovW, ExposureMask);
    XMapWindow(D, ovW);

    for (;;) {
        XEvent  e;
        XNextEvent(D, &e);
        switch (e.xany.type) {
            case Expose: {
                short v[2];
                /* Draw in main planes */
                if (GLXwinset(D, W) < 0) {
                    printf(“winset failed\n”);
                    exit(1);
                }
                reshapeviewport();
                ortho2(0, 1, 0, 1);
                bgnpolygon();
                for (i = 0; i < 4; i++) {
                    c3s(stamp[i].color);
                    v2s(stamp[i].vertex);
                }
                endpolygon();
                /* draw in overlays */
                if (GLXwinset(D, ovW) < 0) {
                    printf(“winset failed\n”);
                    exit(1);
                }
                color(0);
                rectf(0.0, 0.0, 1.0, 1.0);
                color(1);
                rectf(.2, .2, .8, .8);
                color(0);
                rectf(.4, .4, .6, .6);
                break;
            }
/* Die on any keystroke */
        case KeyPress:
            exit(0);
        }
    }
}
```

Chapter 3

Using GLdebug

This chapter describes the GLdebug software tool and tells you how to use GLdebug to assist in debugging graphics applications.

GLdebug Basics

GLdebug helps you locate programming errors in an executable caused by using IRIS GL calls incorrectly. It also shows a graphical representation of the state of the IRIS GL while your program is running. GLdebug helps you visualize what your program is doing and helps you locate usage violations so you can correct your code, but it does not correct the code for you.

GLdebug features:

- a history file for tracing IRIS GL calls made during program execution
- a Stateviewer tool for viewing IRIS GL state during program execution
- a Controller tool for managing debugging sessions interactively

GLdebug verifies IRIS GL state and parameter(s) for every IRIS GL call issued in the program and performs error checking based on those verifications.

State Checking

State checking verifies that the IRIS GL is properly set up to allow the IRIS GL call.

State checking verifies these conditions:

- The IRIS GL has been initialized. The IRIS GL is initialized when you call **winopen()**.

- The IRIS GL call is allowed from the calling location. For example, calling **gconfig()** from within a **bgn/end** structure is not allowed.
- The IRIS GL call is allowed in the current mode and the IRIS GL has been set to the correct mode for the specified operation.

Modes are listed below, with usage examples:

draw mode	Must be set correctly to allow the draw operation. For example, **zdraw()** is allowed in NORMALDRAW mode only
doublebuffer mode	Must be true to call **swapbuffers()**
immediate mode	Must be true to call **qdevice()**
matrix mode	Calling **lmbind()** is not allowed in MSINGLE mode
multimap mode	Must be true to call **setmap()**
pick mode	Calling **viewport()** is not allowed in pick mode
select mode	Must be enabled to allow user selection from screen objects
feedback mode	Must be enabled for hardware feedback operations
rgb mode	Must be true to call **RGBcolor()**

Parameter Checking

Parameter checking verifies that the parameter *value* obeys the following conditions, as defined for each parameter:

- *value* is the required type defined for the parameter. For example, the value must be a device type for **qdevice()**.
- *value* is one member of a set of values defined for the parameter. For example, the value must be one of {MSINGLE, MVIEWING, MPROJECTION, MTEXTURE} for **mmode()**.
- *value* is within the range defined for the parameter. For example, the value must be in the range 0-255 for **RGBcolor()**.
- *value* is not NULL or non-zero. For example, *value* cannot be NULL for **charstr()**.

Error Checking

GLdebug generates error messages based on the results of the state and parameter verification. Errors are categorized according to severity:

Warnings	States or parameter values that are legal but may lead to unpredictable results.
Errors	Illegal operations or illegal parameter values, as defined by the IRIS GL implementation.
Fatals	Errors that result in a core dump and exit the program.

The history file contains a list of every IRIS GL call, warning, error, and fatal error that occurs during program execution.

GLdebug uses these files:

- */usr/lib/libgd_s*
- */usr/sbin/gldebug*
- */usr/sbin/gd_convert*
- */usr/sbin/gd_stateview*
- */usr/sbin/gd_controller*
- */usr/lib/x11/app_defaults/GLdebug*

Running GLdebug

You can run GLdebug from the Admin menu of the CASEVision™/WorkShop Debugger or invoke it separately. See the *CASEVision/WorkShop User's Guide*, Volume I for instructions.

Note: To use GLdebug, your application program must be compiled and linked to the shared IRIS Graphics Library with the *-lgl_s* option. ♦

To invoke GLdebug enter:

`gldebug` [*-gldebug_options*] *executablename* [*-program_options*]

Use *-gldebug_options* to specify the operational behavior of GLdebug and to control the contents of the GLdebug output file(s). Substitute the name of your program and its options for *executablename* and *-program_options*.

Using GLdebug Options

Use these options to select which GLdebug tools to run and to specify the level of error reporting and the type of output to use in the history file:

-h	Do not generate a history file
-w	Suppress warning listings in the history file
-e	Suppress error listings in the history file
-f	Suppress fatal listings in the history file
-c	Do not run the Controller while debugging
-s	Do not run the Stateviewer while debugging
-*C*	Generate the history output in C code
-*F*	Flush the contents of the output buffer to the history file after every IRIS GL call
-p wait	Profile the output
-i filename	Ignore debugging for the IRIS GL calls listed in *filename*
-o filename	Specify the name of the output file as *filename*
-*O*	Send output to the screen (*stdout*)

Specifying History Output

The default filename for the GLdebug output file is *GLdebug.history*. Use the *-o filename* option to specify a name for the history file other than the default. If you prefer to view the IRIS GL calls step-by-step while your program executes, use the -*O* option to direct output to the screen. If both the -*O* and the *-o* options are invoked concurrently, the -*O* option takes precedence and output goes to the screen rather than to the history file.

GLdebug.history contains a listing of every IRIS GL call made by the program, along with the parameter values associated with each call. Parameters used to return values to the program are listed as “OUT”. The return values of functions are not output.

GLdebug.history also lists all warnings, errors, and fatal errors encountered during program execution, unless they are suppressed by an option. Use the *-h* option to disable history output.

Generating History Output as C Code

Use the *-C* option to generate history output in C language. This code can be used to produce a program duplicating the IRIS GL behavior of the original program, but the history file may require additional editing before it will compile. Currently, valid C code is produced for approximately 80% of the IRIS GL calls.

Forcing History Output

Output is buffered to the history file. The contents of the buffer are flushed to the history file at every breakpoint. When a program crashes before the output buffer is completely flushed to the history file, the resulting trace is incomplete. For programs that tend to crash abruptly, you can obtain a complete trace by using the *-F* option, which forces the contents of the output buffer to be flushed to the history file after every IRIS GL call.

Profiling History Output

Use the *-p* option to generate a profile. The profile contains a count of the number of times each IRIS GL subroutine is called. GLdebug writes the count to a file named *GLdebug.count*, according to the value entered for the *wait* parameter. The count is written to the output file after *wait* number of IRIS GL calls have been executed.

Selecting Subroutines to be Written to the History Output

History files that track every IRIS GL call tend to grow large very quickly. To focus the history trace, create a file listing the IRIS GL calls that you wish to be executed, but not written to the history. Enter each IRIS GL call on a separate line in the file. To define the non-traceable IRIS GL calls, specify the name of your file with the *-i* (ignore) option when you invoke GLdebug.

Getting Started with GLdebug

When GLdebug is invoked on a pure IRIS GL program, that is, a program in which **winopen()** is called for window creation, this sequence of events occurs:

1. The graphics application output window appears on the screen.

   If the program allows the user to size the window, use the left mouse button to drag the window corner to the desired size. Otherwise, simply drag the window to the desired location on the screen.

   Nothing appears in the program output window yet, because GLdebug automatically sets a *breakpoint* on the first IRIS GL call. A breakpoint is a statement that is flagged to tell the system to halt the program before executing the statement. Setting breakpoints is discussed in detail under "Using the Controller" on page 96.

2. The Controller window appears on the screen (unless you have disabled it with the *-c* option).

   The Controller window is already set to a fixed size, so simply drag the window to the desired screen location.

3. The Stateviewer appears on your screen (unless you have disabled it with the *-s* option).

When GLdebug is invoked on a *mixed-model* GL/X program (a program that uses X Window System commands as well as GL calls), the Controller and Stateviewer appear directly after the first **GLXlink()** call is made. This typically occurs before any application windows appear. Since a breakpoint is set, execution is halted until you click the *Continue* button.

Using the GLdebug Tools

This section describes the GLdebug Stateviewer and Controller tools.

Using the GLdebug Stateviewer

Stateviewer is a IRIS GL status visualization tool that helps you visualize what is actually happening within the system during program execution. The Stateviewer displays icons that represent IRIS GL conditions. *Options* let you view more detailed information, as described in "Viewing Additional Information from the Options Menu" on page 91.

Figure 3-1 shows the Stateviewer.

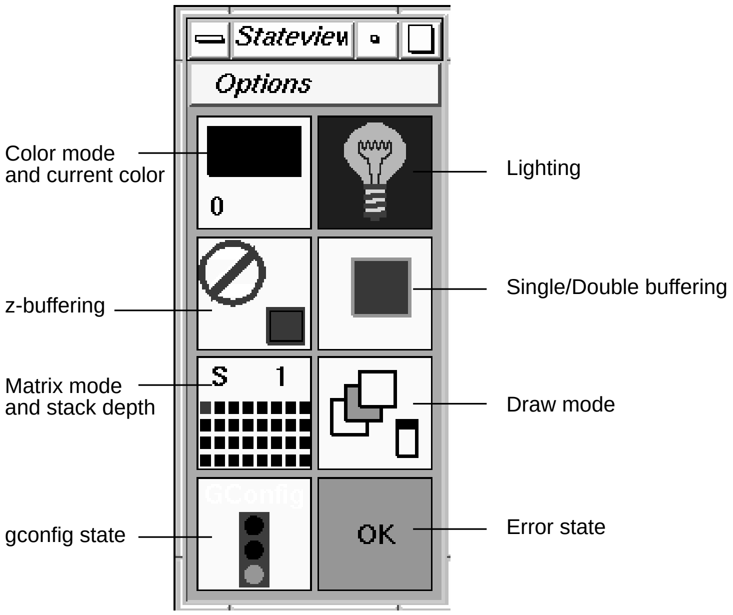


Figure 3-1 Stateviewer

Stateviewer is updated continuously while GLdebug is executing the program, to show the following conditions:

- color mode and current color
- lighting activity
- *z*-buffering activity
- double buffering activity
- matrix mode and depth
- draw mode
- gconfig status
- error status

The Stateviewer icons are described in the following sections.

Color Indicator

Figure 3-2 shows the color indicator.

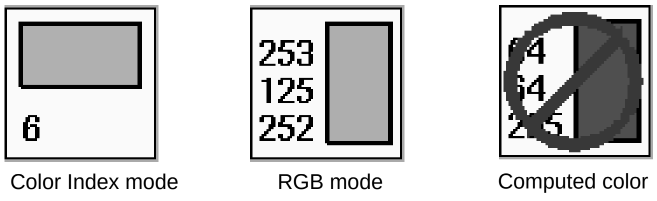


Figure 3-2 Color Indicator

The color indicator displays a color bar and the numerical value(s) of the associated color. In color index mode, the color bar is horizontal and the color value is shown as a single number. In RGB mode, the color bar is vertical and the color value is broken down into its red, green, and blue components.

When the current color in the IRIS GL state is a computed color, for example, from a lighted material vertex, a universal *no* sign appears over the color bar, signifying that the color displayed in the color bar does not accurately represent the color that appears in the image.

Lighting Indicator

Figure 3-3 shows the lighting indicator.

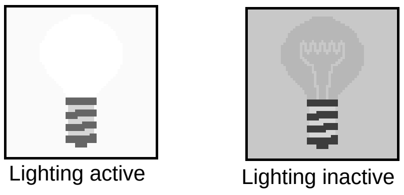


Figure 3-3 Lighting Indicator

The lighting indicator displays a lightbulb that is on when lighting is active; off when lighting is not active. Select *Lighting* from the *Options* menu to view additional lighting information.

Z-Buffer Indicator

Figure 3-4 shows the *z*-buffer indicator.

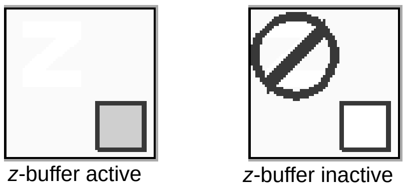


Figure 3-4 *z*-buffer Indicator

The *z*-buffer indicator displays the letter **Z**. A universal *no* sign appears over the **Z** when *z*-buffering is not used. The square shows the *z*-buffer status:

Red square	*z*-buffer is undefined
White square	*z*-buffer is cleared
Yellow square	*z*-buffer is filled
Green outline	*z*-buffer is enabled for drawing (using **zdraw()**)
Red outline	*z*-buffer is disabled for drawing

The *z*-buffer is not enabled for drawing until **zdraw()** is called.

Double Buffering Indicator

Figure 3-5 shows the double buffering indicator.

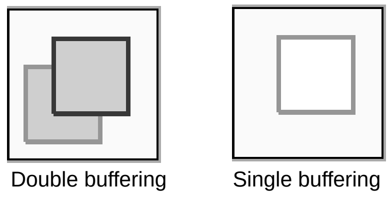


Figure 3-5 Double Buffering Indicator

Two overlapping square icons, representing the front and back buffers, indicate double buffering; one square indicates single buffering. Initially, the front buffer icon appears on top of the back buffer icon. Each **swapbuffers()** call changes the orientation of the buffer icons, as the active buffer moves to the front. The color codes are the same as the *z*-buffer color codes.

Matrix Mode Indicator

Figure 3-6 shows the matrix mode indicator.

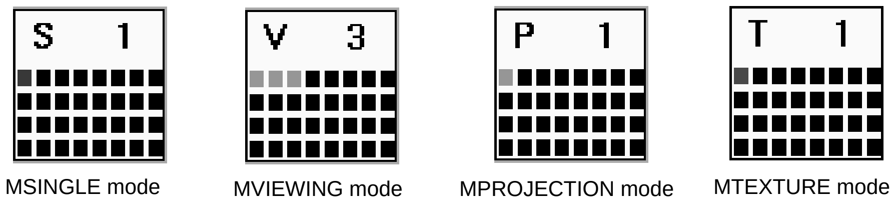


Figure 3-6 Matrix Mode Indicator

The matrix mode (*mmode*) indicator displays the matrix stack depth and the current matrix mode as follows:

MSINGLE mode	**S** represents the single matrix which is used for all modeling, viewing and projection transformations. MSINGLE is the default mode.
MVIEWING mode	**V** represents a multi-matrix mode, where the ModelView matrix is modified by all matrix operations.
MPROJECTION mode	**P** represents the active ModelView and Projection matrices.
MTEXTURE mode	**T** represents the texture option for systems that support textures.

The grid of 32 small squares represents the matrix stack. When a matrix is pushed onto the stack, the square representing the top of the stack turns green. The square turns black when the matrix is popped off the stack. All the squares turn red when there is a stack overflow.

The stack contents are shifted to reflect every **pushmatrix()** and **popmatrix()** operation, so that the green squares always show the depth of the matrix stack. A number in the upper right-hand corner of the indicator also displays the stack depth.

Select *Matrix* from the *Options* menu to view the contents of the current matrix.

Draw Mode Indicator

Figure 3-7 shows the draw mode indicator.

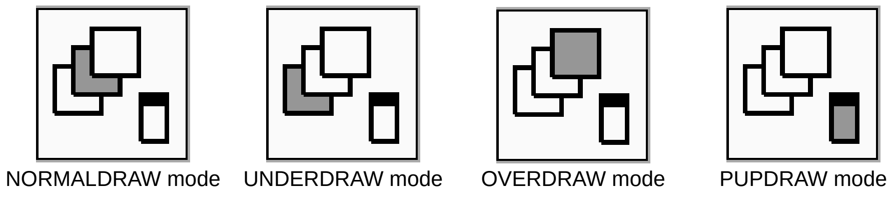


Figure 3-7 Draw Mode Indicator

The draw mode indicator displays a group of square icons that represent drawing surfaces. The active drawing surface appears in green. The position of the square indicates the drawing position and corresponding draw mode:

NORMALDRAW mode	middle square is green
UNDERDRAW mode	bottom left square is green
OVERDRAW mode	top right square is green
PUPDRAW mode	bottom right rectangle is green

GConfig Indicator

Figure 3-8 shows the GConfig indicator.

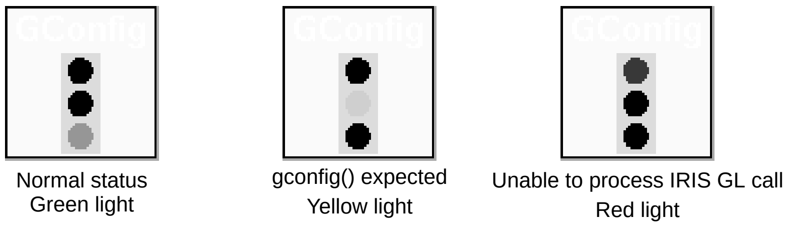


Figure 3-8 GConfig Indicator

The GConfig indicator displays a traffic light icon. Normally, the green light is illuminated.

The yellow light is illuminated when you call a IRIS GL subroutine that requires a subsequent **gconfig()** call. If a subsequent IRIS GL call that depends upon **gconfig()** is executed before **gconfig()** is called, the light changes to red. If **gconfig()** is called first, the light changes back to green.

For example, calling **RGBmode()** causes the light to turn yellow. If **RBGcolor()** is called before **gconfig()**, the light turns red. If **gconfig()** is called before **RGBcolor()**, the light turns green.

The GConfig icon is ignored for mixed-model GL/X windows, because the graphics configuration of such a window is specified at creation time and cannot be changed.

Error Condition Indicator

Figure 3-9 shows the error condition indicator.

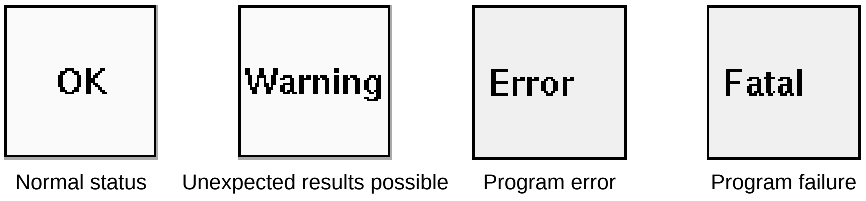


Figure 3-9 Error Condition Indicator

The indicator is green and displays the “OK” message when there are no errors or warnings.

When a warning occurs, the square turns yellow and displays the “Warning” message. The warning indicator remains yellow for the duration of the program, unless an error occurs. When an error occurs, the square turns red and displays the “Error” message, or the “Fatal” message for a fatal error.

You can use the Controller to force execution past a fatal error by setting a breakpoint on “Fatal”, then skipping the fatal statement when the breakpoint occurs. Refer to “Using the Controller” on page 96 for more information about setting breakpoints.

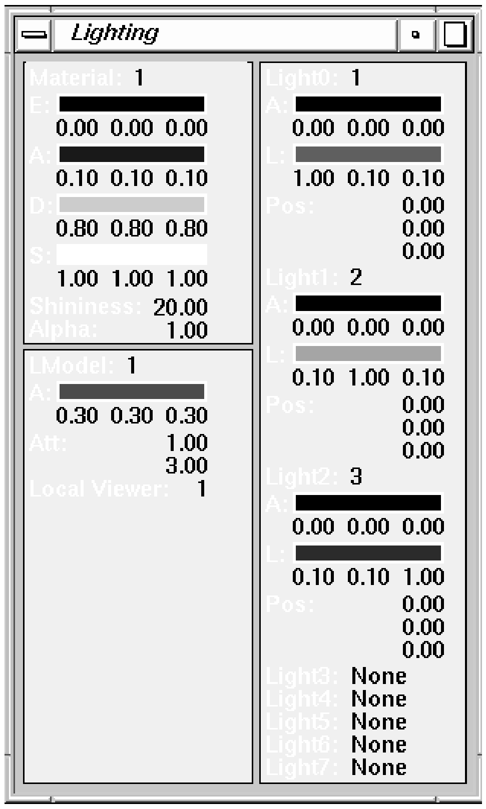
Viewing Additional Information from the Options Menu

Figure 3-10 shows the Options menu. Use the Options menu to view additional IRIS GL status information on lighting, devices, attributes, and the current matrix. Slide the selection bar over the desired item and release the mouse button to display the selected item.

Figure 3-10 Options Menu

Lighting

Figure 3-11 shows the Lighting window.



```
Material:  1                 Light0:  1
E:                           A:
 0.00  0.00  0.00             0.00  0.00  0.00
A:                           L:
 0.10  0.10  0.10             1.00  0.10  0.10
D:                           Pos:             0.00
 0.80  0.80  0.80                             0.00
S:                                            0.00
 1.00  1.00  1.00            Light1:  2
Shininess:  20.00            A:
Alpha:       1.00             0.00  0.00  0.00
                             L:
LModel:  1                    0.10  1.00  0.10
A:                           Pos:             0.00
 0.30  0.30  0.30                             0.00
Att:         1.00                             0.00
             3.00            Light2:  3
Local Viewer:   1            A:
                              0.00  0.00  0.00
                             L:
                              0.10  0.10  1.00
                             Pos:             0.00
                                              0.00
                                              0.00
                             Light3:  None
                             Light4:  None
                             Light5:  None
                             Light6:  None
                             Light7:  None
```

Figure 3-11 Lighting Window

The Lighting window displays attributes for each of the defined lighting models. Material, lighting model, and light source attributes are displayed for each lmdef number that is currently bound. When you activate a material, a lighting model, or a light by calling **lmbind()**, the active attributes are displayed. If an attribute is not bound, "None" is displayed for that value.

Material properties are in the upper left corner of the Lighting window. Emission and reflection properties are shown as color bars with numeric values; shininess and alpha values are listed. Material properties are:

E	Indicates the emission property of the material. The emission value is represented by three RGB values. The color and intensity of the emitted light is shown in the bar.
A	Indicates the ambient reflection property of the material.
D	Indicates the diffuse reflection property of the material.
S	Indicates the specular reflection property of the material.
Shininess	Indicates the shininess property of the material.
Alpha	Indicates the alpha index of the material.

Lighting model (Lmodel) properties are shown directly below the material properties. The ambient light characteristic is shown as a color bar with the associated ambient coefficients listed below it. Attenuation and local viewer assignments are also listed. Lmodel properties are:

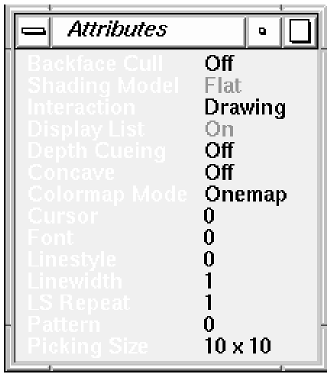
A	Indicates the ambient characteristic of the lighting model.
Att	Lists the attenuation factors for the lighting model. The top number represents the k_0 coefficient and the bottom number represents the k_1 coefficient.
Local	Indicates the status of the Local Viewer flag. A value of “1”
Viewer	Indicates that the flag is set, a “0” indicates that the flag is not set.

Light source properties are listed for each light that has been bound with **lmbind()**. Color bars show the ambient and intensity properties of each light. Light source properties are:

A	Indicates the ambient light characteristic of the light source.
L	Indicates the intensity characteristic of the light source.
Pos	Indicates the position of the light source in *x,y,z* coordinates.
Dir	Indicates the direction of the light source in *x,y,z* coordinates (*not shown*).

Attributes

Figure 3-12 shows the Attributes window.



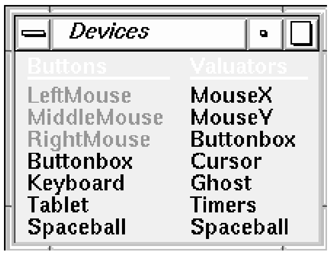
```
Backface Cull    Off
Shading Model    Flat
Interaction      Drawing
Display List     On
Depth Cueing     Off
Concave          Off
Colormap Mode    Onemap
Cursor           0
Font             0
Linestyle        0
Linewidth        1
LS Repeat        1
Pattern          0
Picking Size     10 x 10
```

Figure 3-12 Attributes Window

The Attributes window displays information about IRIS GL attributes. Attributes are displayed with their default values in black. If the program changes any of the attributes, the new value is displayed in green.

Devices

Figure 3-13 shows the Devices window.



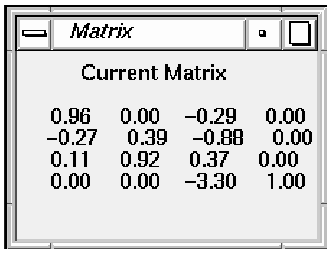
Buttons	Valuators
LeftMouse	MouseX
MiddleMouse	MouseY
RightMouse	Buttonbox
Buttonbox	Cursor
Keyboard	Ghost
Tablet	Timers
Spaceball	Spaceball

Figure 3-13 Devices Window

The Devices window displays a list of buttons and valuators that can be used for input to the program. When a device is queued, its name is displayed in green. This list should be ignored for GL/X windows, because it reflects the state of IRIS GL input devices, not X input devices.

Matrix

Figure 3-14 shows the Matrix window. The Matrix window displays the values in the current transformation matrix.



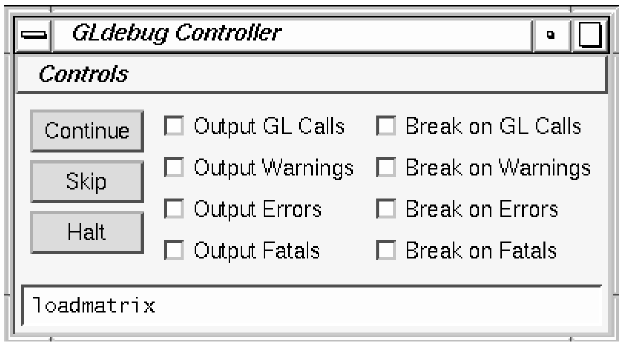
```
      Current Matrix

 0.96   0.00  -0.29   0.00
-0.27   0.39  -0.88   0.00
 0.11   0.92   0.37   0.00
 0.00   0.00  -3.30   1.00
```

Figure 3-14 Current Matrix

Using the Controller

Controller is an interactive debugging tool that lets you select the level of debug output and lets you control program execution.

Figure 3-15 shows the Controller.



```
loadmatrix
```

Figure 3-15 Controller

The Controller contains a Controls menu, control buttons, two sets of toggle buttons for enabling outputs and breakpoints, and a message window. The next sections describe how to operate these features.

Selecting Actions from the Controls Menu

Figure 3-16 shows the Controls menu.

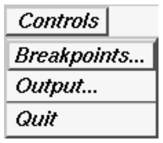


Figure 3-16 Controls Menu

The Controls menu lets you access windows for setting breakpoints, for specifying output, and for quitting GLdebug:

Breakpoints...	Displays a list of IRIS GL calls that can be selected as breakpoints.
Output...	Displays a list of IRIS GL calls that can be selected for output to the history file.
Quit	Exits GLdebug and closes all associated windows.

Using the Control Buttons

The control buttons control program execution in debug mode. Use the left mouse button to activate the desired control button:

Continue	Resume program execution after a breakpoint.
Skip	Resume program execution at the statement following the breakpoint. The breakpoint statement itself is not actually executed when Skip is selected. This feature can be used to skip over fatal errors and examine program behavior beyond the failure point.
Halt	Forces a breakpoint at the next IRIS GL call, whether or not it has been previously defined as a breakpoint.

The message window at the bottom of the Controller displays the IRIS GL call where the program is halted. The IRIS GL call shown in the message window is the next statement to be executed.

Setting Breakpoints and Output

Breakpoints are used to freeze program execution at selected statements. Breakpoints let you examine the program output at a particular instant in time, much like viewing a still frame from a movie.

Breakpoints are typically set on errors so that the behavior of the program prior to the occurrence of the error can be examined. Execution of the program is halted just before the IRIS GL call selected as the breakpoint executes, but after a trace of this call is written to the history file.

Use the Controls Menu to access the Output and Breakpoint selection windows. These windows display a list of IRIS GL calls.

Figure 3-17 shows the Breakpoint and Output selection windows.

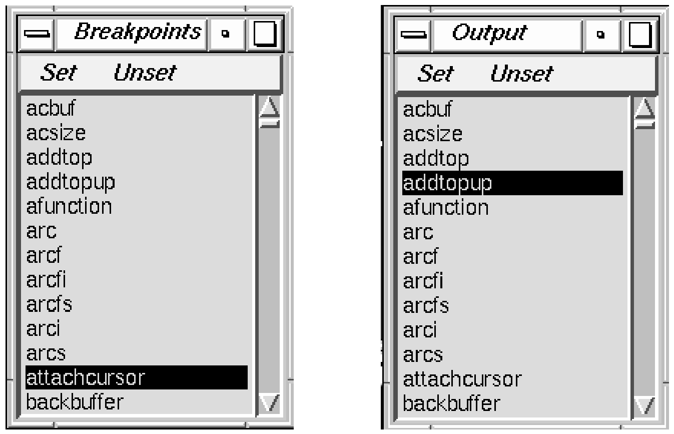


Figure 3-17 Breakpoint and Output Selection Windows

Initially, all IRIS GL calls are selected for both output and breakpoints. Selected functions appear in inverse video (white text on a black background). You can streamline your debugging by focusing the output and the breakpoints on suspected problem areas. Select or deselect the desired IRIS GL calls individually with the mouse, or by group from the Set or Unset pulldown menus.

To select an individual IRIS GL call from the list, click the left mouse button on the desired call. Select a block of IRIS GL calls by holding down the mouse button and dragging. If you drag past the top or bottom of the window, the list of calls continues to scroll.

You can also select a block by clicking on the first item in the block and then shift-selecting, holding down the **`<Shift>`** key while clicking on the last item. Select two or more non-adjacent items by holding down the **`<Control>`** key while clicking left. Control-selecting a previously selected item deselects it.

Summary of mouse selection actions:

Click left	Select an item.
`<Control>` + click	Select additional non-adjacent item(s) or deselect previously selected item(s).
`<Shift>` + click	Select a block of items, from the first item selected up to and including the last item selected.
Click + Drag	Continue selecting items, and scroll window until mouse button is released or until end of list is reached.

Use the Set and Unset pulldown menus to select from the following functional groups of IRIS GL calls:

All	Selects every IRIS GL call listed.
Swapbuffers	Selects the **swapbuffers()** call.
Geometry	Selects all IRIS GL calls associated with geometry.
Transforms	Selects all IRIS GL calls associated with matrix operations.
Lighting	Selects all IRIS GL calls associated with lighting.
Objects	Selects all IRIS GL calls associated with display list objects.
Texturing	Selects all IRIS GL calls associated with texture operations.
GL Input	Selects all IRIS GL calls associated with performing user input.
Text	Selects all IRIS GL calls associated with text operations.
None	Deselects all the IRIS GL calls.

Set enables output/breakpoints for a functional group. Unset deselects the group.

For example, choosing Texturing from the Set menu selects all IRIS GL subroutines associated with texturing, as shown in Figure 3-18.

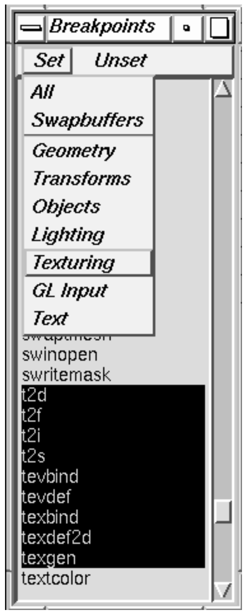


Figure 3-18 Using the Set Menu to Set Breakpoints on Texturing

Using the Toggle Buttons

The Output and Break toggle buttons let you enable/disable outputs and breakpoints for IRIS GL calls, Warnings, Errors, and Fatal errors. The default state is all toggles on.

There is an Output toggle and a Break toggle for each of the following:

- GL Call
- Warning
- Error
- Fatal

The toggle button must be on to enable the selected outputs and breakpoints. If the toggle buttons are off, no output or breakpoints will occur, regardless of what is selected from the Output and Breakpoint windows.

Use the left mouse button to operate the toggle buttons. A toggle is on when it is light gray in color, appears to be pushed in, and shows a yellow LED. A toggle is off when it is dark gray, appears to be pushed out, and does not show an LED.

When you are finished debugging your application, select Quit to exit GLdebug and to quit your application.

Chapter 4

Tuning IRIS GL Applications

The process of analyzing software performance and adjusting the code to obtain improved performance is known as *tuning*. Just as tuning up a car makes it run more efficiently, tuning code makes the software run faster and facilitates optimum use of hardware capabilities.

This chapter presents programming techniques and a methodology to help you achieve maximum performance from your IRIS GL application.

The techniques described in this chapter are also incorporated into Silicon Graphics's IRIS Performer™ software development environment. IRIS Performer automatically optimizes graphical applications on the full range of IRIS products without changes or recompilation. Performance features supported by IRIS Performer include:

- data structures to use the CPU, cache, and memory system architecture efficiently
- tuned rendering loops to convert the system CPU into an optimized data management engine
- state-management control to minimize overhead

For more information on IRIS Performer, contact your sales representative.

Tuning Basics

This section outlines the basic tuning strategy and presents techniques for taking timing measurements and for isolating performance problems. It presents a three-stage model of the IRIS Geometry Pipeline® and explains pipeline tuning considerations.

Why is Tuning Useful?

Effective code tuning is a natural part of graphics programming. A clear, consistent programming style combined with a systemized approach to performance tuning produces efficient, understandable, and maintainable code. Tuning need not detract from software readability and modularity. This chapter provides you with a conceptual framework to guide you in the tuning process and details specific techniques that are effective and easy to implement.

Because the hardware of Silicon Graphics systems is so fast, you might think that you won't be able to make a significant difference in perceived performance by tuning your code. This is not the case, because in addition to the graphics capabilities of your system, the speed at which an application runs is a limiting factor—even the fastest machine can only render as fast as the application can drive it. Simple changes in application code can make a dramatic difference in rendering time.

There are two quick changes that give your program a big performance advantage, especially if you are running older IRIS GL code, which does not use the newer vertex subroutines (for example, **v3f()**). For code that uses old-style subroutines, adding the following line may make your program go as much as 30% faster:

```
glcompat(GLC_OLDPOLYGON,FALSE);
```

On newer graphics subsystems, such as PowerVision™ (IRIS-4D/VGX™), including this line can double the performance of some programs:

```
subpixel(TRUE);
```

These mode settings should be used by most programs on recent Silicon Graphics workstation models; however, to preserve backwards compatibility they are not the default settings, so you must set them in your program.

Three-Stage Model of the Graphics Pipeline

For software tuning purposes, the graphics pipeline in all Silicon Graphics workstations may be modeled by three operational stages:

1. The application program running on the CPU, feeding commands to the graphics subsystem.
2. Per-polygon operations, such as coordinate transformations, lighting, depth-cueing, clipping, and concave polygon decomposition.
3. Per-pixel operations, such as the simple operation of writing color values into the frame buffer, or more complex operations such as *z*-buffering, alpha blending, and texture mapping.

Figure 4-1 shows the three-stage model of the graphics pipeline.

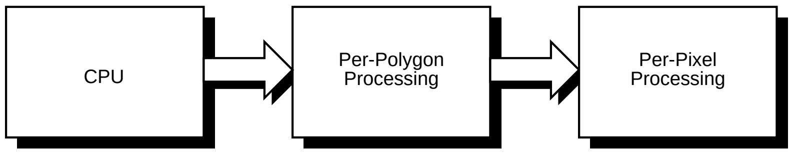


Figure 4-1 Three-Stage Model of the Graphics Pipeline

This three-stage model is simpler than the actual hardware implementation in the various models in the Silicon Graphics product line, but it is detailed enough for all but the most subtle tuning tasks.

The amount of work required from the different pipeline stages varies among applications. For example, consider a program that draws a small number of large polygons. Because there are only a few polygons, the pipeline stage that does per-polygon operations is lightly loaded. Because those few polygons cover many pixels on the screen, the pipeline stage that does per-pixel operations is heavily loaded. To speed up this program, you must speed up the per-pixel stage, either by drawing fewer pixels or by drawing pixels in a way that takes less time, by turning off modes like texturing, blending, or **z**-buffering.

Furthermore, because spare capacity is available in the per-polygon stage, you can increase the work load at that stage without degrading performance. For example, you can use a more complex lighting model, or define geometries such that they remain the same size but look more detailed because they are composed of a larger number of polygons.

Pipeline Tuning

Traditional software tuning focuses on finding and tuning *hot spots*, the 10% of the code in which the program spends 90% of its time. *Pipeline tuning* uses a different approach — rather than looking for hot spots, you look for *bottlenecks*, overloaded stages that are holding up other processes, in the pipeline.

Imagine a factory assembly line. The station on the assembly line that takes the longest to complete a task limits the speed of the entire line. No one downstream can process more work, because the downstream flow is limited by the output of the slowest station. No one upstream can process more work, because the line backs up and slows down to match the slowest station. The only way to get more performance out of the entire assembly line is to speed up the operation of the slowest task.

A pipeline-oriented graphics system behaves in a similar way. At any given time, one stage of the pipeline is the limiting factor — the bottleneck. Reducing the time spent in the bottleneck is the only way to improve performance. Speeding up operations in other parts of the pipeline has no effect. Conversely, doing work that further intensifies the bottleneck, or that creates a new bottleneck somewhere else, is the only thing that will further degrade performance. The work load can be increased at other parts of the pipeline without degrading performance, as long as it does not become so slow that it becomes a new bottleneck. In this way, an application can sometimes be altered to draw a higher quality image with no degradation of performance.

Different programs stress different parts of the pipeline, so it’s important to understand which elements in the graphics pipeline are the bottlenecks for your program.

The following parameters determine the performance of most applications:

- total number of polygons in a frame
- transform rate for the given polygon type and mode settings
- number of pixels filled
- fill rate for the given mode settings
- time of screen and/or *z* clears
- time to swap buffers
- time of application overhead

Isolating Bottlenecks in the Pipeline

The basic strategy for isolating bottlenecks is to measure the time it takes to execute a program or part of a program and then change the code in ways that don't alter its performance except by adding or subtracting work at a single point in the graphics pipeline at a certain time. If changing the amount of work processed at a given stage of the pipeline does not alter performance appreciably, that stage is not the bottleneck. If there is a noticeable difference in performance, you've found a bottleneck.

Finding CPU Bottlenecks

The most common bottleneck occurs when the application program does not feed the graphics subsystem fast enough. Such programs are called *CPU-limited*.

To test for a CPU bottleneck, you want to remove as much graphics work as possible, while preserving the behavior of the application in terms of the number of instructions executed and the way memory is accessed. Often, changing just a few IRIS GL calls is a sufficient test. For example, replacing vertex and normal calls like **v3f()** and **n3f()**, with color subroutines, like **c3f()**, preserves the CPU behavior while eliminating all drawing and lighting work in the graphics pipeline. If making these changes does not significantly improve performance, then your application has a CPU bottleneck.

Finding Per-Polygon Bottlenecks

Programs that create bottlenecks in the per-polygon stage are called *transform-limited*. To test for bottlenecks in per-polygon operations, change the program so that the application code runs at the same speed and the same number of pixels are filled, but the per-polygon work is reduced. For example, if you are using lighting, temporarily turn lighting off. Bind your material to 0 and see if performance improves. If performance improves, then your application has a per-polygon bottleneck.

Finding Per-Pixel Bottlenecks

Programs that cause bottlenecks at the per-pixel stage in the pipeline are *fill-rate limited*. To test for bottlenecks in per-pixel operations, shrink objects, or shrink the window size to reduce the number of active pixels. This technique won't work if your program alters its behavior based on the sizes of objects or the size of the window. You can also reduce the work done per pixel by turning off per-pixel operations such as *z*-buffering or alphablending. If any of these experiments speeds up your program, then your application has a per-pixel bottleneck.

Break Complex Programs Into Smaller Pieces

Many programs draw a variety of different things, each of which stresses different parts of the system. Decompose such a program into pieces and time each piece. You can then focus your bottleneck isolation and tuning activities on the slowest pieces.

Taking Timing Measurements

Timing, or *benchmarking*, a piece of graphics code requires some care.

Follow these steps to get accurate timing measurements:

1. Take measurements on a quiescent system.

   Verify that no unusual activity is taking place on your system when you take timing measurements. Other graphics programs, background processes, and network activity can distort your timing results.

2. Choose timing trials that are not limited by the clock resolution.

   If your system updates its clock at intervals of one-hundredth of a second, and the phenomenon you're trying to measure lasts only one one-hundredth of a second, the expected error percentage of any given timing trial is very large. A good rule of thumb is to benchmark something that takes at least two seconds, so that the uncertainty contributed by the clock reading is less than one percent of the total error. To measure something that is faster than two seconds, write a loop to execute the test code repeatedly.

3. Benchmark static frames.

   Verify that the code you are timing behaves identically for *each frame* of a given timing trial. If the scene changes, the current bottleneck in the graphics pipeline may change, making your timing measurements meaningless. For example, if you are benchmarking the drawing of a rotating airplane, choose a single frame and draw it repeatedly, rather than letting the airplane rotate while you are taking the benchmark. Once a single frame has been analyzed and tuned, you can look for frames that stress the graphics pipeline in different ways, then analyze and tune them individually.

4. Call **finish()** before reading the clock at the start and at the end of the time trial.

   Graphics calls can be tricky to benchmark because they do all their work in the graphics pipeline. When a program running on the main CPU issues a graphics command, the command is put into a hardware queue in the graphics subsystem, to be processed whenever the graphics pipeline is ready to process it. The main CPU can immediately go on to do other work, including issuing more graphics commands until the queue fills up.

   When benchmarking a piece of graphics code, you must include in your measurements the time it takes to process all the work left in the queue after the last graphics call. To accomplish this, call **finish()** at the end of your timing trial, just before sampling the clock. Call **finish()** before sampling the clock and starting the trial, to make sure no graphics calls remain in the graphics queue ahead of the process you are timing. Sample benchmarking code is provided in Appendix A, "Benchmarking Tools."

Tuning to Frame Rates

The smoothness of an animation depends on its *frame rate*. The more frames per second, the smoother the motion appears. Smooth animation also requires double buffering. In double buffering, one frame buffer holds the current frame, which is scanned out to the monitor by video hardware, while the rendering hardware is drawing into a second buffer that is not visible. When the new frame buffer is ready to be displayed, the system swaps the buffers. The system must wait until the short vertical retrace period between raster scans to swap the buffers, so that each raster scan displays an entire stable frame, rather than parts of two or more frames.

Therefore, frame rates must be integer multiples of the screen refresh time, which is 16.7 milliseconds (msec) for a 60 Hz monitor. If the draw time for a frame is slightly longer than the time for *n* raster scans, the system waits until the *n+1st* vertical retrace before swapping buffers and allowing drawing to continue, so the total frame time is *(n+1)*16.7* msec. *Thus, a change in the time spent rendering a frame has no visible effect unless it changes the total time to a different integer multiple of the screen refresh time*. If you want an observable performance increase, you must reduce the rendering time enough to take a smaller number of 16.7 msec raster scans. Alternatively, if performance is acceptable, you can add work without reducing performance, as long as the rendering time does not exceed the current multiple of the raster scan time.

Follow these steps to optimize frame rate performance:

1. To produce an observable performance increase, reduce drawing time to a lower multiple of the screen refresh time.
2. Take timing measurements in single buffer mode only.

   To get accurate numbers, you must perform timing trials in single buffer mode, with no calls to **swapbuffers()**. Because buffers can only be swapped during a vertical retrace, there is a dead period, between the time a **swapbuffers()** call is issued and the next vertical retrace, when a program may not execute any graphics calls. If a program attempts to issue graphics calls during this period, it is put to sleep until the next vertical retrace, distorting the accuracy of the timing measurement.

3. Perform non-graphics computation after **swapbuffers()**.

   A program is free to do non-graphics computation during the wait cycle between vertical retraces. Therefore issue a **swapbuffers()** call immediately after sending the last graphics call for the current frame, perform computation needed for the next frame, then execute IRIS GL calls for the next frame, call **swapbuffers()**, and so on.

"Three-Stage Model of the Graphics Pipeline" on page 105 introduced a three-stage model of the graphics pipeline, consisting of CPU, per-polygon and per-pixel operations. The following sections assume that you know which part of the pipeline you are trying to optimize. They describe techniques for writing and tuning graphics code for a particular stage of the graphics pipeline.

Suggestions for graphics programming for peak performance are also described. However, writing high-performance code is usually more complex than just following a set of rules. Most often, it involves making trade-offs between special functions, quality, and performance for a particular application. After reading these sections, experiment with the different techniques described to help you decide where to make these trade-offs.

CPU Tuning

When an application is CPU-limited, the entire graphics pipeline may be sitting idle for periods of time. This section describes techniques for structuring application code so that the CPU doesn't become the bottleneck.

To get the best possible CPU performance, follow these two overall guidelines:

1. Compile your application for optimum speed.

   Compile all object files with at least *-O2*. Note that the compiler option for debugging, *-g*, turns off all optimization. If you must run the debugger on optimized code, you can use *-g3* with *-O2* with limited success. For faster floating point operations, compile with *-float*.

2. Use a simple data structure and a fast traversal method.

   The CPU tuning strategy focuses on developing fast database traversal for drawing with a simple, easily accessed data structure. The fastest rendering is achieved with an inner loop that traverses a completely flattened (non-hierarchical) database. Most applications cannot achieve this level of simplicity for a variety of reasons. For example, some databases occupy too much memory when completely flattened.

Suggestions for database structure and traversal methods that allow you to manage and balance the performance/memory conflict follow.

Optimizing Cache and Memory Use on IRIS POWER Series Systems

This section discusses efficient use of cache and memory in IRIS POWER Series™ multiprocessor architectures. These architectures use MIPS® R2000® and R3000® series processors. Refer to "Optimizing Cache and Memory Use on IRIS Crimson IP17 Processors" on page 115 for information about the IRIS Crimson IP17 processors.

Most systems do not have an unlimited amount of fast memory. To approach this ideal, system memory is structured as a hierarchy that contains a small amount of faster, more expensive, memory at the top and a large amount of slower memory at the base.

The hierarchy is organized from registers in the CPU at the top, down to the disks at the bottom. As memory locations are accessed, they are automatically copied into higher levels of the hierarchy, so data that is accessed most often is placed in the fastest memory locations.

The two levels of the memory hierarchy with which you should be most concerned are the *cache*, which feeds data to the CPU and the *translation-lookaside buffer* (TLB), which keeps track of frequently used pages of memory.

Each processor has an instruction cache and two data caches. The purpose of the caches is to feed data and instructions to the CPU at maximum speed. When data is not found in the cache, a *cache miss* occurs and a performance penalty is incurred as data is brought into the cache.

Locations in virtual memory are found from a page table that translates virtual pages to physical memory. This page table can be very large and can be paged itself, therefore, such lookups can become very expensive. The solution to this problem is to *cache the translations of the most frequently used pages* in the TLB. The TLB is fully associative, that is, a page may be placed in any location on the TLB. The only restriction is the limit on the number of pages and there are currently 56 page entries available to user processes. If a page translation is not found in the TLB, a delay is incurred to look up the page and enter its translation.

The goal of machine designers and programmers is to maximize the chance of finding data in a top level of the hierarchy. To achieve this goal, algorithms for maintaining the hierarchy, encoded into the hardware and the operating system, assume that programs have *locality of reference* in both time and space, keeping frequently accessed locations close together. You get increased performance by respecting the degree of locality required by each level in the memory hierarchy.

Even applications that appear not to be memory intensive, in terms of total number of memory locations accessed, may suffer unnecessary performance penalties for inefficient allocation of these resources. An excess of cache misses, especially misses on *read* operations, can force the most optimized code to be CPU-limited. Memory paging causes almost any application to be severely CPU-limited.

Follow these guidelines to minimize memory paging:

1. Keep frequently used data within a minimal number of pages (each page consists of 4K (4096) bytes).

   Minimize the number of pages accessed in your program by keeping data structures within as few pages as possible and verify that TLB misses are not occurring. Instructions for tracking TLB activity are provided later in this section.

2. Minimize cache misses.

   Each processor has a first-level data cache of 64KB and a second-level data cache of 256KB. The first-level data cache is actually a subset of the data in the second-level cache. The data caches are direct-mapped, so the location of the data in the cache depends exclusively on its lower address bits. A *cache line* is a block of four 32-bit words. On a read-miss, a block of four full cache lines is brought into the cache.

Structure your data so the most frequently accessed data remains in the first-level cache wherever possible.

Locate data within cache lines to get the maximum benefit from each cache miss. Each cache miss brings sixteen 32-bit words into the cache. If you're accessing words sequentially, each cache miss brings in sixteen words of needed data; if you're accessing every sixteenth word, each cache miss brings in one needed word and fifteen unneeded words, degrading performance by up to a factor of sixteen.

Avoid conflicts where data in two separate cache lines map to the same location in the cache. If data structures access multiple pages, or are very large, conflicts for cache locations may occur, causing additional misses. For large, packed arrays this is unavoidable. However, the impact can be minimized by packing each cache line with currently needed data.

Avoid simultaneously traversing two large buffers of value data, such as an array of address flags and an array of **cpack()** data, because there can be cache conflicts between the two buffers. Instead, pack the contents into one buffer.

Second-level data cache misses also increase bus traffic, which can be a problem in a multi-processing application. Three processors missing their second-level cache will saturate the MP bus that connects the CPU, memory, and graphics subsystems. This can happen with multiple processes traversing very large data sets.

3. Minimize page (TLB) misses.

   The same concerns exist for page access as for cache access. If you access words sequentially, each TLB miss brings in a TLB entry to be used for the next page of 1024 32-bit words. If you access words that are 4096 bytes apart, every access results in a TLB miss.

4. Measure cache-miss and page-fault overhead.

   To find out if cache and memory usage are a significant part of your CPU limitation:

   - Use **osview** to monitor your application.

     You should not see any page faults or *tfaults* - TLB misses. To quickly estimate the effect of memory access for just graphics calls, compare the running time of your rendering loop with graphics calls completely stubbed out to the time with graphics calls left in, but with **v3f()** calls turned into **c3f()** calls. When graphics calls are completely stubbed out, no graphics data is accessed or sent to the pipe. When graphics calls are left in, with **v3f()** calls changed to **c3f()** calls, all data is accessed but the graphics subsystem doesn't spend any time drawing.

- A more rigorous way to estimate the time spent on memory access is to compare the results obtained by *PC sampling* vs. *basic block counting* analyses for the two different runs, with and without **v3f()**'s. PC sampling, from *prof(1)*, gives a real-time estimate of the time spent in different sections of the code. Basic block counting, from *pixstats(1)*, gives an ideal estimate of how much time should be spent, not including memory references. See the *prof(1)* and *pixstats(1)* man pages for information on these tools. PC sampling includes time for system overhead, so it always predicts longer execution than *pixstats* basic block counting. However, your PC sample time should not be more than 1.5 times the time predicted by *pixstats*. For detailed information on profiling, see Chapter 4, "Improving Program Performance," in the *IRIX System Programming Guide*, which you can read online from the IRIS InSight viewer.

These experiments help you determine if the significant problem is waiting on cache, TLB misses, or page faults for your graphics data.

Optimizing Cache and Memory Use on IRIS Crimson IP17 Processors

IRIS Crimson™ systems contain an IP17 CPU board with the MIPS R4000® CPU. The MIPS R4000 CPU architecture differs from the MIPS R3000 processors used in Silicon Graphics POWER Series systems in ways that may have performance implications for graphics applications.

The Crimson CPU provides a significant increase in instruction rate over earlier system architectures. Crimson delivers a 100 MHz instruction rate (compared with 40 MHz in R3000-based systems). This rate brings previously CPU-limited graphics programs closer to achieving peak graphics performance.

The Crimson CPU has a direct-mapped unified 1 MByte secondary cache for both instructions and data. Therefore, much larger data sets fit in the secondary cache on a Crimson system than do on POWER Series systems, for example, the 200 and 300 series systems. However, data sets may experience conflicts with instructions in the unified secondary cache. Try to minimize cache conflicts, as they can force frequently needed data out of the cache.

The Crimson CPU uses a different method for transferring data to the graphics subsystem than do the POWER Series CPUs. For optimal graphics performance on the 200, 300, and 400 POWER Series systems, it is important that graphics geometry data for IRIS GL commands that send information to the graphics subsystem by address, such as **v3f()**, **n3f()**, **c3f()**, and **t2f()**, be kept in memory that is allocated separately from other program data. The reason for this is that on these systems, geometry data is transferred to the graphics subsystem without going through the cache.

On Crimson systems, geometry data is sent to the graphics subsystem through the CPU, so it is needed in the cache. Therefore, separating graphics geometry data from other data is not necessary on a Crimson system; in fact, doing so may have a negative effect on performance because it increases the probability of cache conflicts.

To achieve peak graphics performance, maximize the chance that a piece of data needed by the CPU, including any type of graphics data on a Crimson system, is in the cache when it is needed. You can use the same techniques described in "Optimizing Cache and Memory Use on IRIS POWER Series Systems" on page 112, except you should not separate your value data from your geometric reference data as recommended for IRIS POWER Series systems.

Tuning Immediate Mode Drawing

Immediate mode provides flexibility and control over both storage management and drawing traversal. The trade-off for the extra control is that you have to write your own subroutines for data traversal. These subroutines, and the data structures they access, must be optimized.

Optimizing Data Structures

The suggestions presented in this section for creating data structures and traversals concern general cache and memory efficiency. Some machines, such as the POWER Series machines, can yield extra performance if some additional constraints are followed.

It is common for scenes to have hierarchical definitions. Scene management techniques may rely on specific hierarchical information. Caching behavior is often difficult to predict for hierarchical dynamic data structures. Hierarchical structures present several performance drawbacks:

- The time spent traversing pointers to different sections of a hierarchy can create a CPU bottleneck.

  This is partly because of the number of extra instructions executed, but it is also a result of the inefficient use of cache and memory. Overhead data not needed for rendering is brought through the cache and can push out needed data, causing subsequent cache misses.
- Traversing hierarchical structures can cause excessive memory paging.

  Hierarchical structures can be distributed throughout memory. It is difficult to be sure of the exact amount of data you are accessing and where it is located, therefore traversing hierarchical structures can access a costly number of pages.
- Complex operations may need access to both the geometric data and other scene information, complicating the data structure.

For these reasons, hierarchy should be used with care. In general, store the geometry data used for rendering in static, contiguous buffers, rather than in the hierarchical data structures. In addition, because Silicon Graphics processors have a direct-mapped cache, avoid having pieces needed for a given object occupying the same locations in the cache where possible.

Follow these steps to optimize data structures:

1. Minimize data structure hierarchy.

   Flatten your rendering data (minimize the number of levels in the hierarchy) as much as cache and memory considerations and your application constraints permit.

2. Separate value data from geometric reference data on POWER Series systems.

   Note: Do *not* separate the value data from geometric reference data on Crimson IP17 systems. The Crimson CPU uses a different method for transferring data to the graphics subsystem than do the POWER Series CPUs. ♦

   The reason for this separation is related to how the POWER Series systems transfer geometric data to the graphics pipe. The high performance graphics subroutines use Direct Memory Access (DMA) to transfer blocks of reference data from memory directly to the graphics pipe without that reference data ever going through cache.

   Whenever value data is pulled into cache, it is brought in as full cache-lines of data (four words on a POWER Series). Therefore, if value data, such as **cpack()** data, or your own command flags, is mixed in with reference data, reference data is pulled into the cache unnecessarily. This effectively destroys locality of reference, wastes valuable cache locations, and increases the likelihood of a future cache-miss occurring on the next piece of value data, that could have been brought in, instead of reference data, with previous value data.

   The "Optimizing Database Traversal" on page 120 discusses ways to organize different types of data for a fast traversal that satisfies this constraint.

3. Use quad-word alignment on POWER Series systems.

   All data sent to high performance graphics subroutines, such as **v3f()**, should begin on a *quad-word address*, an address divisible by 16. Using quad-word alignment is simple. A quad word is four words, so always allow the data for a given command four 32-bit words of space. For example, an integer or a float takes up one word, therefore **c3f()**, **n3f()**, and **v3f()** data for one 3-D vertex uses three words. Instead of packing the data for each command, always leave a fourth word as padding and start the data for the next command at the next quad-word.

   The following example shows four component color data alternating with vertex data:

   ```
   r g b a x y z _ r g b a x y z _
   ```

   In the case of normal and color data, each gets a word of padding:

   ```
   r g b _ x y z _ ....
   ```

The need for quad-word alignment depends on how the CPU gets data from memory to the graphics system. On the POWER Series machines, such as the IRIS-4D/GTX or VGX, the DMA hardware uses a 3-way handshake between CPU, memory and graphics to send data to the graphics subsystem. Blocks of data are sent directly to the graphics pipe without ever going through cache.

Each bus transaction consists of a quad-word that begins on an address that is divisible by 16. Thus a call to a subroutine, like **v3f()**, with quad-word aligned data requires a single bus transaction. Two transactions are required for data that is not quad-word aligned. Therefore, unaligned polygons without other significant bottlenecks, can suffer a performance loss in comparison with quad-word aligned data. If a traversal is otherwise CPU-limited, or there is a bottleneck in the graphics pipeline, then changing to quad-word alignment may not have a noticeable effect.

Quad-word alignment increases the size of your graphics data. However, because geometry data does not go through cache, you may only notice the effect from this increase if you are accessing too many pages, causing additional TLB misses or actually running out of memory.

Note: IRIS GL display lists automatically quad-word align all data sent to graphics subroutines. ♦

The following sample C language code fragment page-aligns a **malloc**-ed buffer, which is also quad-word aligned, because page boundaries are on quad-word boundaries.

```
/* malloc with a page of overhead */
pointer = (float *) malloc (num_floats*sizeof(float) + 4096);
/* bump up the pointer a page */
pointer = (float *) ((int) pointer) + 4095);
/* All page addresses are divisible by 4096 = 2^12 so zeroing
 * the bottom 12 bits will put us back to the previous page
 * boundary which is automatically a quad word boundary.
 */
pointer = (float *) (((int) pointer) & 0xfffff000);
```

4. Use *floats* or *ints*, but not *shorts*, on VGX systems.

   On a VGX system, the subroutines for sending data to the graphics pipe as *shorts*, for example **c3s()**, and **v3s()**, are less efficient than the other subroutines, because the VGX does not always use the DMA method for data transfer for these commands. Therefore, using *shorts* on a VGX can be CPU-limiting.

Optimizing Database Traversal

This section includes some suggestions for writing peak-performance code for inner rendering loops.

Ideally, an application should be spending most of its time in database traversal and sending data to the graphics pipe. Instructions in the display loop are executed many times every frame, creating hot spots. Any extra overhead in a hot spot is greatly magnified by the number of times it is executed.

When using simple, high-performance graphics primitives, the application is even more likely to be CPU-limited. The data traversal must be optimized so that it does not limit the speed of all drawing.

Follow these steps to optimize data traversal:

1. Use peak-performance code for drawing subroutines.

   During rendering time, the sections of code that actually issue graphics commands should be the hot spots in application code. These subroutines should use peak-performance coding methods. Small improvements to a line that is executed for every vertex in a database accumulate to cause a noticeable effect when the entire frame is rendered.

   Follow these suggestions for writing peak-performance code:

   - Use single-dimensioned arrays for data.
   - Use flat data structures and do not use multiple pointer indirections when rendering:

     ```
     bad:    v3f(object->data->vert);
     ok:     v3f(dataptr->vert);
     best:   v3f(dataptr);
     ```

The following sample code fragment is an example of efficient code to output a single gouraud, lit, polygon from quad-word aligned data. Notice that a single data pointer is used. It is updated once at the end of the polygon, after the **endpolygon()** call.

```
bgnpolygon();
n3f(ptr);
v3f(ptr+4);
n3f(ptr+8);
v3f(ptr+12);
n3f(ptr+16);
v3f(ptr+20);
n3f(ptr+24);
v3f(ptr+28);
endpolygon();
ptr += 32;
```

- Do not have short, fixed-length loops, especially around vertices. Instead, unroll these loops:

```
bad:
for(i=0; i < 4; i++){
    cpack(poly_colors[i]);
    v3f(poly_vert_ptr[i]);
}

good:

cpack(poly_colors[0]);
v3f(poly_vert_ptr[0]);
cpack(poly_colors[1]);
v3f(poly_vert_ptr[1]);
cpack(poly_colors[2]);
v3f(poly_vert_ptr[2]);
cpack(poly_colors[3]);
v3f(poly_vert_ptr[3]);
```

- Minimize the work done in a loop to maintain and update variables and pointers. Unrolling can often assist in this:

```
bad:     n3f(*(ptr++); v3f(*(ptr++));
or       n3f(ptr); ptr += 4;
         v3f(ptr); ptr += 4;
good:    n3f(*(ptr)); v3f(*(ptr+1)); n3f (*(ptr+2));
         v3f(*(ptr+3));
or       n3f(ptr); v3f(ptr+4); n3f(ptr+8); v3f(ptr+12);
```

- Minimize the number of different buffers accessed in a loop:

```
bad:   n3f(normaldata); t2f(texdata); v3f(vertdata);
good:  n3f(dataptr); t2f(dataptr+4); v3f(dataptr+8);
```

- Use simple *switch* statements instead of multiple *if-else-if* control structures.
- Avoid division. Shift or multiply by a reciprocal rather than perform integer divides.
- Prototype subroutines in ANSI C style to avoid run-time typecasting of parameters:

```
void drawit( float *ptr, int count)
{
  .......
}
```

- Avoid typecasting of values, which happens at run-time:

```
val = (float) *ptr;
```

  Typecasting of pointers occurs at compile time and is efficient:

```
long *ptr;
*(float *) ptr = float_val;
float_val = *(float *) ptr;
```

- Make end-conditions on loops as trivial as possible, for example, compare the loop variable to a constant, preferably 0. Therefore, decrementing loops are often more efficient than their incrementing counterparts:

```
bad:     for (i=0, i*size < (end - begining)/size; i++)
           {...}
better:  for (i +beginning, i < end; i +=size)
           {...}
good:    for (i=total, i > 0; i--)
           {...}
```

- Do not do *if* tests around vertices—use duplicate code instead. An example of this is shown under rule 2 on page 123.

2. Use specialized drawing subroutines and macros.

   Decide how geometry should be displayed at as high a level in the program organization as possible. The drawing subroutines should be highly specialized leaves in the program tree. Decisions made too far down the tree can be redundant. For example, consider a program that switches back and forth between drawing flat-shaded and gouraud-shaded. Once this choice has been made for a frame, the decision is fixed and the flag is set. For example, the code illustrated below is extremely inefficient:

```
/* Inefficient way to toggle modes */
draw_object(float *data, int npolys, int gouraud)  {
int i=npolys;
for (; i > 0; i--) {
    bgnpolygon();
    if (gouraud) c3f(data);
    v3f(data + 4);
    if (gouraud) c3f(data +8);
    v3f(data + 12);
    if (gouraud) c3f(data + 16);
    v3f(data + 20);
    if (gouraud) c3f(dta + 24);
    v3f(data + 28);
    endpolygon();
}
```

   Even though the choice of drawing mode was made before the draw_object routine was entered, the flag is checked for every vertex in the scene. A simple *if*-test may seem innocuous; however, when done on a per-vertex basis, it can accumulate a noticeable amount of overhead.

   Compare the size and number of useless NOPs (no operations) in the disassembled code for the routine, first without, and then with, the *if*-test.

Assembly code for routine without *if*-tests (19 instructions):

```
jal  bgnpolygon
nop
jal  n3f
move a0,s0
jal  v3f
addiu  a0,s0,16
jal   n3f
addiu   a0,s0,32
jal  v3f
addiu  a0,s0,48
jal  n3f
addiu  a0,s0,64
jal  v3f
addiu  a0,s0,80
jal  n3f
addiu  a0,s0,96
jal  v3f
addiu  a0,s0,112
jal  endpolygon
```

Assembly code for routine with *if*-tests (27 instructions):

```
jal  bgnpolygon
nop
beq  s1,zero,0x3c
nop
jal  n3f
move  a0,s0
jal  v3f
addiu  a0,s0,16
beq  s1,zero,0x54
nop
jal  n3f
addiu  a0,s0,32
jal  v3f
addiu  a0,s0,48
beq  s1,zero,0x6c
nop
jal  n3f
addiu  a0,s0,64
jal  v3f
addiu  a0,s0,80
beq  s1,zero,0x84
nop
```

```
jal  n3f
addiu  a0,s0,96
jal  v3f
addiu  a0,s0,112
jal  endpolygon
```

Notice how many extra instructions precede each **n3f()** in the *if*-test code. That extra *if*-test per-vertex increases the number of instructions executed for this otherwise optimal code by 40%. For code that is otherwise less optimal, the effect can still be significant as a result of swapping data in registers or other unpredictable interactions. These effects may not be visible if such code is used only to render objects that are always graphics limited. However, if the process is CPU-limited, then moving decision operations, such as this *if*-test, higher up the in the program structure, improves performance.

The appropriate place to check the flag for gouraud shading is *outside* the loop. To do this, create two specialized subroutines: one routine for gouraud shading and one for flat shading. Redundancies in writing such duplicate code can usually be effectively managed with macros.

3. Preprocess drawing data

   Putting some extra effort into generating a simpler database makes a world of difference when traversing that data for display. A common tendency is to leave the data in a format that is good for loading or generation of the object, but not optimal for actually displaying it. For peak performance, you should do as much of the work as possible *before* rendering.

   Preprocessing turns a difficult database into a database that is easy to render quickly. This is typically done at initialization or in changing from a modeling to a fast-rendering mode. For example, consider a database that has many-sided or concave polygons. Many sided, or concave, polygons are very slow to render. You can break those polygons into quads and triangles, which can then be grouped into qstrips and/or tmeshes, for sorting of like-primitives.

   Preprocessing can also be used to turn general meshes into fixed-length strips.

The following sample code shows a commonly used, but very inefficient, way to write a tmesh render loop.

```
while (!done) switch(*data) {
    case BGNMESH:
        bgntmesh();
        break;
    case ENDMESH:
        endtmesh();
        break;
    case SWAPMESH:
        swaptmesh();
        break;
    case EXIT:
        done = 1;
        break;
    default: /* have a vertex !!! */
        n3f(dataptr);
        v3f(dataptr + 4);
        dataptr += 8;
}
```

This type of traversal incurs a significant amount of per-vertex overhead. The loop is evaluated for every vertex and every vertex must also be checked to make sure that it is not a flag. This wastes time and also brings all of the object data through the cache. This practice reduces the performance advantage of using meshes. Any variation of this code that has per-vertex overhead is CPU-limiting on the peak graphics machines, like the VGX, and even more so on the CPU-limited IRIS-4D/85GT™, for most types of simple graphics operations.

This possibility also applies to another popular tmesh coding scheme, the vertex loop:

```
bgntmesh();
for (i=num_verts; i > 0; i--) {
    n3f(dataptr);
    v3f(dataptr+4);
    dataptr += 8;
    }
endtmesh();
```

For peak immediate mode performance, the best way to optimize this code is to precompile meshes into specialized primitives of fixed length strips. Only a few fixed lengths are needed. For example, use strips of length 12, 8, and 2.

These specialized meshes are then sorted by size, resulting in the efficient loop shown in this sample code:

```
/* dump out N 8-triangle meshes */
for (i=N; i > 0; i--) {
    bgntmesh();
        n3f(dataptr);
        v3f(dataptr+4);
        n3f(dataptr+8);
        v3f(dataptr+12);
        n3f(dataptr+16);
        v3f(dataptr+20);
        /* now each new vertex is a new tri */
        n3f(dataptr+24);
        v3f(datatpr+28);
        ...
    endtmesh();
    dataptr += 80;;
}
```

The unrolling of the drawing code, illustrated above, can be made more compact with specialized macros like these:

```
#define LIT_GOURAUD_VERT(ptr, offset) \
    {n3f(ptr + offset); v3f(ptr + offset + 4);}
#define  LIT_GOURAUD_MESH_LENGTH_8        (dataptr)\
    {\
    LIT_GOURAUD_VERT(dataptr, 0);    \
    LIT_GOURAUD_VERT(dataptr, 8);\
    .....
    LIT_GOURAUD_VERT(dataptr, 72); \
    }
```

A mesh of length 12 is about the maximum for unrolling. Unrolling helps to reduce the overall cost-per-loop overhead, but after a point, it produces no further gain. Over-unrolling eventually hurts performance if done to an extreme in several places.

Pre-compilation of complex data enables the peak-performance techniques described above and is often necessary for some of the specialized techniques described below.

4. Experiment and benchmark potential traversals

   While the techniques in this section might not be used in their entirety, this underlying strategy should be applied:

   - Minimize the CPU work done at the per-vertex level.
   - Use a simple data structure for the rendering traversal.

Writing a peak-performance immediate mode renderer for a specific application can be an interesting and challenging task. Unfortunately, there is no recipe for this sort of task. Design potential data structures and traversal loops and write small benchmarks that mimic the memory demands you expect in order to predict the CPU-limitation of your traversal. Experiment with small optimizations and benchmark the effects. Experimentation on small examples can save time in the actual implementation.

Sample Data Structure and Traversal

The data structure in Example 4-1, *perfobj*, can be used in the optimal case, as well as in the more dynamic and complex situations. This example shows a very simple, but complete and general structure. Most applications only use a small subset of this structure. Some applications may add specializations.

Example 4-1 The *perfobj* Data Structure

```
/* structure to organize geometry data */
typedef struct perfobj_vert_t {
    float vertex[4];  /* padded for quad-word alignment */
    float color4[4];
    float normal[4];
    float texture[4];
} perfobj_vert_t;

typedef struct perfobj_t {
    /* data buffers - page align these */
    float *flags;              /* DL flags and value data */
    perfobj_vert_t *vdata;     /* for geometry data */
    float *props;              /* props for lmdef, texdef */
    unsigned long *texdata;    /* texdef */
    Matrix mdata[PD_MAX_MATRICES]; /* matrix data */
    /* keep track of the size of all data */
    int flags_nlongs;
    int vdata_nlongs;
    int props_nlongs;
    int texdata_nlongs;
    int mdata_nmats;
} perfobj_t;
```

The *perfobj* data structure has a separate buffer for value data. This buffer contains the immediate mode display-list flags and associated value data. For example, you can have a command flag called GOURAUD_QUAD and have the flag followed by a pointer to the corresponding vertex data in the geometry data buffer. Corresponding **c3f()** and **n3f()** data can be located in the *vdata* structure. Value data, such as **cpack()** data, can be stored with the flags.

Other data buffers contain geometric data, matrix data, property data for IRIS GL items, such as lights, materials, textures, and image data. A strict sizing account of these structures is kept in the **_nlongs* fields. This organization represents convenient classes of usage, cache flow, size, and type. The geometric reference data is kept separately from other data in the *vdata* structure. The *perfoj_vertex_t* structure organizes the geometric data and keeps it quad-word aligned. All geometric data is of the type *float*.

The *perfobj* structure becomes the basic building block for the application's data structures. The data structure is customized with specialized command flags put in the *flags* array for use by the traversal. Simple applications can put their entire scene in a single perfobj. Complex applications may have many *perfobj*s for a scene and perhaps an additional *perfobj* for global information. The *flag*s array for the global *perfobj* may then contain specialized command flags that include jumping to additional *perfobjs*.

The *flags* array should contain commands that encompass as many actions as possible. The traversal can then loop over a *switch* statement for the flags. The following example is an excerpt from such a traversal:

```
/* perfobj command handler */
switch(*flagsP) {
....
case TREE:
    /* trees need to be oriented and translated */
    pushmatrix();
    translate (*(flagsP+1), *(flagsP+2), *(flagsP+3));
    rot(*(flagsP+4),’z’);
    cpack(*(flagsP+5)); /* base tree color */
    /* macro to dump out flat-shaded, textured, quad */
    DRAW_FLAT_TEX_QUAD(flagsP+6);
    flagsP+=7;
    break;
case GOURAUD_QUAD_8: /* blast out 8 gouraud quads */
    DRAW_GOURAUD_QUAD(flagsP+1);
    DRAW_GOURAUD_QUAD(flagsP+2);
    ....
    DRAW_GOURAUD_QUAD(flagsP+8);
    flagsP+=9;
    break;
case LIT_TMESH_4: /* blast out lit tmesh of 4 tris */
    bgntmesh();
    n3f((perfobj_vert_t *) (flagsP+1)->normal);
    v3f((perfobj_vert_t *) (flagsP+1)->vertex);
    ...
    n3f((perfobj_vert_t *) (flagsP+6)->normal);
    v3f((perfobj_vert_t *) (flagsP+6)->vertex);
    endtmesh();
    break;
    ....
}
```

Tuning Display Lists

Display lists simplify drawing because you don’t have to optimize your own traversal of the data. In addition, display lists manage their own data storage, which is particularly nice for algorithmically generated objects. Another advantage of display lists is that they are significantly better for remote graphics over a network. This is because the display list traversal can be done on the remote hardware. Also, assuming no display list editing is done, the display list can be cached on the remote CPU so that the data for the display list does not have to be re-sent every frame.

Display lists do have some drawbacks that may affect some applications, the most troublesome of these is *data expansion*. To achieve a fast, simple, traversal on all machines, all data is copied directly into the display list. Therefore, the display list contains an entire copy of all your data and, for each command, a jump address that is an extra four bytes. All geometric data is quad-word aligned. For vertex and color data alone, this adds an extra 33-66% to the data size: a jump address plus the four long word coordinates for every vertex command (**v3f()**, **c4f()**, **n3f()**, and so on). This is in addition to the application storage requirements.

Furthermore, many vertices are stored repeatedly. In a grid mesh, each vertex is used in four quads, so it is stored four times. An arbitrary mesh can be even worse. If the database becomes significantly large, paging eventually hinders performance. Therefore, when considering the use of IRIS GL display lists, consider cache size and, for extremely large databases, the amount of main memory.

Tuning for display lists focuses mainly on reducing storage requirements. The ability of the data to fit in cache improves performance because it avoids cache-misses as the data is retraversed.

Note: Crimson IP17 systems maximize performance for IRIS GL display lists, including very large display lists that do not fit in the cache. ♦

Follow these steps to optimize display list drawing:

1. Call **delobj()** to delete objects no longer used.

   This frees storage space used by the deleted objects and expedites the building of new objects and object rendering. If there are serious memory shortages, call *chunksize()* to control the size of each **malloc()**. If there are many small objects, use smaller **malloc()**s to reduce wasted space. If there are large objects, use large **malloc()**s to help reduce memory fragmentation.

2. Avoid duplication of objects.

   For example, if you have a scene for 100 spheres of different sizes, generate one object that is a unit sphere centered about the origin. For each sphere in the scene:

   - Set the material for the current sphere.
   - Issue the necessary scaling factors for sizing the sphere and the transformations for positioning the sphere.
   - Issue a **callobj()** to the unit sphere.

   In this way, a jump address for the unit sphere is stored instead of storing all of the sphere vertices for each instance of the sphere.

3. Generate an entire object at one time, rather than frequently opening and closing it, to avoid memory fragmentation.

4. Make the display list as flat as possible.

   Avoid using an excessive hierarchy with many **callobj()** calls. Each **callobj()** call requires an extra set of computation for the new object and several independent objects can cause memory fragmentation and/or paging. A flat display list requires less memory and yields a simpler and faster traversal.

Display lists are best used for static objects. Even the most trivial display list editing is extremely costly. Do not put dynamic data or operations in display lists. Instead, use a mixture of display lists for static objects and immediate mode for dynamic operations.

Advanced CPU-limited Tuning Techniques

This section lists some advanced techniques for tuning CPU-limited code.

Use these advanced techniques to tune CPU-limited applications:

1. Mix computation with graphics.

   When you are fine-tuning an application, interleaving computation and graphics can create a balanced, and therefore more efficient, system. Key places for interleaving are after **swapbuffers()**, **clear()**, **zclear()**, and known fill-limited drawing.

The **swapbuffers()** call creates a special situation. After issuing a call to **swapbuffers()**, an application may be forced to wait, in the worst case, up to 16.7 msecs for the next vertical retrace before issuing more graphics calls. For a program drawing at 10 frames per second, 15% of the time (worst case) can be spent waiting for the **swapbuffers()** to occur.

In contrast, non-graphic computation is not forced to wait for a vertical retrace. Therefore, if there is a section of computation that must be done every frame that includes no graphics calls, it can be done here instead of causing a CPU-limitation somewhere else.

On the IRIS-4D/GT or Personal IRIS™ graphics systems, clearing the screen is an expensive operation and doing non-graphics computation immediately after the clear is more efficient than sending more graphics down the pipeline when it isn't ready, then doing the non-graphics computation in a place where the application is CPU-limited.

Experimentation is required to:

- Determine where the application is reliably graphics limited.
- Ensure that inserting the computation does not create a new bottleneck.

For example, if the new computation references a large section of data that is not in the data cache, the data for the drawing may be swapped out for the computation, then swapped back in to draw again, which may result in worse performance than the original organization.

2. Use *dis* or *compile* with *-S* to look at assembly code

   When tuning inner rendering loops, looking at assembly code can be extremely helpful. Use *dis* with the *-p* option to specify a procedure, to disassemble optimized code for a given procedure, and correlate assembly code lines with line numbers from the source code file. This is especially helpful for examining optimized code. The *-S* option to *cc* produces a *.s* file of assembly output, complete with your original comments.

   You need not be an expert in MIPS assembly code to interpret the results. Just looking at the number of extra instructions required for what looks like an innocuous operation is very informative. Knowing some basics about MIPS assembly code can be helpful for finding performance bugs in inner loops. See *MIPS RISC Architecture*, by Gerry Kane, for additional information.

3. Use an additional processor for complex scene management.

   If your application is running on systems with multiple processors, consider supplying an option for doing scene management on additional processors to relieve the rendering processor from the burden of expensive computation.

   Additional processors may also be used to reduce the amount of data rendered for a given frame. Simplifying and/or reducing rendering for a given scene can help reduce bottlenecks in all parts of the pipeline, as well as the CPU. One example is removing unseen or backfacing objects. Another common technique is to use an additional processor to determine when objects are going to appear very far away and use a simpler model with fewer polygons and less expensive modes for distant objects. This is known as *level-of-detail* rendering.

Tuning Transform-Limited Drawing

This section presents techniques that you can use to tune applications that are transform-limited.

Using Fast Drawing Modes

Use **subpixel(TRUE)** and **shademodel(FLAT)** whenever possible.

On a VGX, calling **subpixel(TRUE)** often doubles performance. The *only* time **subpixel(TRUE)** can slow you down is when using Personal IRIS or GT graphics with RGB antialiased lines.

Calling **shademodel(FLAT)** significantly improves CPU, transform, and fill-limited performance on all machines. Machines that take special advantage of flat-shading include the VGX, the SkyWriter™, IRIS Indigo™, and the Personal IRIS. On the Personal IRIS and IRIS Indigo flat-shading is especially recommended for high-performance lines.

Using High-Performance Drawing Subroutines

Use high performance graphics subroutines such as **v3f()** with bgn/end constructs for meshes, polygons, lines and points. Use **n3f()** instead of **normal()** and **c3f()** or **cpack()** instead of **RGBcolor()**.

Old library subroutines are slower than the high-performance subroutines for several reasons. First, old-style polygons are automatically filled and then given an outline for compatibility reasons. This outline is needed only in rare instances (concave polygons or t-junctions) where "cracking" between polygons can occur at certain angles. Turning on outlining to eliminate t-junctions is very costly and does not always work.

Therefore, any program that uses old-style polygon subroutines, such as **polf()** or **pdr()**, should call **glcompat(GLC_OLDPOLYGON,FALSE)** before drawing to turn off outlining. This greatly reduces the work done per polygon, and thereby yields an immediate improvement in the drawing rate. Across the product line, high-performance drawing subroutines are typically about 30% faster than old-style drawing subroutines. Especially avoid the incremental old style subroutines: **pdr()**, **pmv()**, **move()**, and **draw()**, which require significant overhead to maintain the current status of each open primitive.

The high performance IRIS GL subroutines are always more efficient than the old-style subroutines, even if outlining is turned off. This is because of a fundamental difference in functionality and implementation. The old style subroutines are general primitives in the true sense of the word: every command is uniquely tied with a primitive. Therefore, a **pdr()** adds a vertex to an open polygon and a **draw()** adds a vertex to an open line. For example, the following code segment gives you a polygon, a line and a character string:

```
pdr();
pmv();
move();
cmov();
draw();
charstr();
draw();
pdr();
pclose();
```

It is expensive to store all the current information for an open primitive to support this kind of generality. For this reason, **pmv()** is much slower than **polf()**. The **polf()** subroutine receives all of its data at once and does not need to maintain an open polygon.

The following code draws the same picture and is much clearer:

```
cmov();
charstr();
move();
draw();
draw();
pmv();
pdr();
pdr();
pclose();
```

The high performance subroutines take advantage of this clearer organization and set drawing modes, as opposed to opening primitives. Once you set a drawing mode, vertex, normal, and color subroutines, such as **v3f()**, **n3f()**, and **c3f()**, cause the appropriate behavior for the current mode. This simplification also allows for more efficient methods of sending data to the graphics subsystem, which reduces the CPU operations for each command. This type of orthogonal organization also eliminates the need for specialized subroutines, thus making the IRIS GL more compact and easy to extend.

Using Peak Performance Primitives for Drawing

This section describes how to draw geometry with optimal primitives.

Follow these steps to optimize drawing:

1. Use connected primitives or quads.

   Connected primitives are very desirable because they reduce the amount of data sent to the graphics subsystem and also reduce the amount of per-polygon work done in the pipeline. Typically, about 12 primitives are required in a bgn/end sequence to achieve peak rates. For lines and points, it is especially beneficial to put as many vertices as possible in a bgn/end sequence. For example, on a VGX, a transform-limited bgn/end line with four vertices is twice as fast as one with two vertices.

2. Use “well behaved” polygons—small and convex with only three or four vertices.

   Polygons with five or more vertices are likely to be transform-limited and are at best only a fraction of the speed of four-sided polygons. Drawing concave polygons can be prohibitively expensive, as much as 1/10 the speed of three- or four-sided convex polygons. Triangle meshes do not need to be checked for concavity, so they never incur a performance penalty for concave testing. Concave checking, by default, is set to FALSE.

   There are some special differences with the GT graphics subsystem. On a GT graphics subsystem, concave breakup is always done, so calling **concave(FALSE)** currently has no effect. This has the additional implication that polygons with more than 4 sides are even more costly as a result of concave testing.

   If your database has polygons that are not well-behaved, write code to perform an initial one-time pass over the database to transform the troublemakers into well-behaved polygons and use the new database for rendering. For the newer machines, such as the VGX and Personal IRIS, using meshed primitives results in additional gains.

3. Call only vertex operations in a bgn/end sequence.

   Within a bgn/end sequence, the only legal graphics commands that may be used are commands for setting colors, normals, texture coordinates, and vertex coordinates. The use of any other graphics calls is illegal and may have unpredictable results. This is specified in Appendix A of the *Graphics Library Programming Guide* and in the man pages for the bgn/end commands, such as **bgnpolygon()**. Even if such calls appear to “work”, such use is not guaranteed to work in the future and may cause severe performance penalties.

4. Minimize the data sent per vertex.

   Polygon rates can be directly affected by the number of normals or colors sent per polygon. With CPU-limited drawing, setting a color or normal per vertex, regardless of the **shademodel()** used, may be slower than setting only a color per polygon, because of the time spent sending the extra data and resetting the current color.

The GT calculates lighting only at each normal, so fewer normals per polygon may speed up lighting because fewer calculations are done per polygon. This is not the case on the VGX, which lights every vertex.

The number of normals and colors per polygon also directly affects the size of a display-list containing the object.

Optimizing Lighting Performance

The IRIS GL offers a large selection of lighting features, some being virtually "free" in terms of computational time, others offering sophisticated effects with some performance penalty. The penalties some features carry may vary between machines and performance of certain features may be greatly improved in the future. In complex situations, you should be prepared to experiment with the lighting configuration.

You normally won't notice a performance degradation when using one infinite light, unless you use lit textures or color index lighting.

Use the following settings for peak performance lighting:

- single infinite light
- RGBmode
- LOCALVIEWER set to 0.0 in the lighting model, the default
- TWOSIDE set to 0.0 in the lighting model, the default
- lmcolor mode set to LMC_COLOR, the default, or LMC_NULL
- **nmode(NAUTO)**, the default

Lighting Operations With Noticeable Performance Costs

Minimize these operations to achieve peak lighting performance:

1. Using multiple lights.

   Multiple infinite lights have a performance penalty for each additional light. This is most severe on the GT graphics subsystem, where multiple lights have a significant penalty.

On most other systems, the performance penalty is much less than it is on the GT, decreasing linearly with the number of additional lights. Most newer systems exhibit only a linear performance degradation for multiple lights.

2. Changing materials or material properties frequently.

   Frequently changing materials, or material properties, can be very expensive. If the current material needs to be changed many times per frame, define a single material that can have specific properties altered, rather than calling **lmbind()** for separate materials many times per frame. There are several ways to do this.

   If the material needs to be changed every polygon or every vertex, use the routine **lmcolor()** to set an lmcolor mode, such as LMC_AD, and avoid frequently changing the lmcolor mode.

   If only a few material properties need to be changed every few polygons, but still many times per frame, you can use the following technique:

```
/* quick lmcolor change */
lmcolor(LMC_AD);
cpack(material_color);
lmcolor(LMC_COLOR);
```

   This retains the peak lighting configuration for the actual drawing of polygons.

   Note: lmcolor() changes are not reflected in the material *definition*. If the material is rebound, it will have its original properties. ♦

   Finally, if many material properties only need to be changed a few times a frame, then you can have a special property array and use **lmdef()** to re-define the currently bound material. For example:

```
static float red_props[] = {
    AMBIENT, 1.0, 0.0, 0.0,
    DIFFUSE 1.0, 0.0, 0.0,
    LMNULL};
    ....
    lmdef(DEFMATERIAL, mat_index, 0, red_props);
```

The graphics pipe currently holds only one current lighting material. When a new material is bound with **lmbind()**, the entire material is loaded. However, when a current material has one or two properties (except SHININESS) re-defined, then only the new properties must be loaded. In this case only, **lmdef()** can be faster than **lmbind()**.

The quick **lmcolor()** change is even faster than **lmdef()**, because it does not update the material database and sends only the changing property to the graphics pipeline.

Lighting Operations With Significant Performance Costs

The features described in this section are classified as having "significant" costs because any one of them, when added to high-performance lighting, causes a substantial drop in performance. However, while adding additional significant features still has some cost, the additional penalty is small compared to the initial drop from high-performance lighting.

Follow these recommendations to limit your use of lighting features that significantly limit performance:

1. Use significant features with care.

   For a minimal performance drop, set LOCALVIEWER to 1.0 in the lighting model, while using infinite lights only. Each additional local light adds noticeable penalties to the transform rate.

   Two-sided lighting lights both sides of a polygon. This is much faster than the alternative of drawing polygons twice to get this effect. However, using two-sided lighting is significantly slower than high-performance lighting for a single rendering of the object. Call **getgdesc(GD_LIGHTING_TWOSIDE)** to determine if two-sided lighting is available on your system.

2. Use unit-length normals.

   Always use unit-length normals when possible. The feature **nmode(NNORMALIZE)** sets to unit length all normals, with a potentially significant performance cost. You can avoid such situations with precompilation. Also, avoid using viewing transformations that result in non-uniform scaling because that forces renormalization of all normals in the graphics pipeline.

Practices to Avoid

Follow these rules when using lighting:

1. Avoid using lighting calls inside a bgn/end sequence.

   If possible, avoid calls to any of the lighting commands, **lmcolor()**, **lmbind()**, or **lmdef()**, during a bgn/end drawing sequence, as this has a very serious performance penalty. While making such calls to change colors, by changing material properties, is technically legal, the performance penalty is severe. This is a special case of the warning to call only vertex operations in a bgn/end sequence discussed in "Using Peak Performance Primitives for Drawing" on page 136.

2. Do not change material SHININESS values.

   Avoid changing the SHININESS value of a material or loading materials with different SHININESS values. This is because each time a new SHININESS value is set, significant computation is required.

Using Expensive Modes Efficiently

The IRIS GL offers many features that create sophisticated effects with excellent performance. However, these features are associated with some performance cost, compared to drawing the same scene without them. The implication is that these features should be used where their effects, performance, and quality are justified.

1. Turn off expensive features when they are not required.

   Once expensive features have been turned on, they can slow the transform rate of vertices even when they have no visible effect. For example, if a texture map is currently bound, then the transform rate of polygons with no texture coordinates is still degraded. Texturing should be explicitly turned off with **texbind(TX_TEXTURE_0,0)** for nontextured polygons.

   The use of fog can slow the transform rate of polygons even when the polygons are too close to show fog, and even when the fog density is set to zero. Explicitly turn off fog with the call **fogvertex(FG_OFF, 0)** for these conditions.

Finally, turn off polygon screen subdivision with the call **scrsubdivide(SS_OFF, 0)**. Screen subdivision is a mode that should be avoided in high-performance applications.

2. Minimize expensive mode changes and sort operations by the most expensive mode.

   Avoid changing texture maps and frequently toggling texture on and off.

   Avoid changing the projection matrix, or **lsetdepth()** parameters, which affect the world-z to z-buffer mapping.

   When fog is on, avoid changing the fog density.

   Turn fog off for rendering with a different projection, for example, orthographic, and turn it back on when returning to the normal projection.

   Expensive mode changes should be minimized. Sort drawing to minimize the most expensive mode changes and beware of excessive changes to any modes, even the cheaper mode changes such as **shademodel()**, **zbuffer()** and **blendfunction()**.

Advanced Transform-limited Tuning Techniques

This section describes advanced techniques for tuning transform-limited drawing.

Follow these hints to draw objects with complex surface characteristics:

1. Use texture to replace complex geometry.

   Textured polygons are significantly slower than their non-textured counterparts. However, texture can be used instead of extra polygons to add detail to a geometry. This can greatly simplify a geometry, resulting in a net speed increase and an improved picture, as long as it does not cause the code to become fill-limited. The texture pixel rate varies across the product line, so this technique might not be as effective for other systems, such as the IRIS Indigo, so experimentation is advised.

2. Use **afunction()**, in conjunction with texture, to give the effect of very complex geometry on a single polygon.

An image of a complex object is textured onto a single polygon. Alpha values are set to zero outside the polygon. Calling **afunction(0, AF_NOTEQUAL)** causes pixels in the polygon with zero alpha values to not be drawn. The edges of the object can be antialiased by using alpha values between zero and one. Finally, the polygon is oriented toward the viewer. This effect is often used to create objects like trees that have complex edges and/or that have many holes through which the background should be visible.

Tuning Fill-Limited Drawing

This section presents techniques for tuning fill-limited drawing.

Using Backface/Frontface Removal

Backface and frontface removal should be used to reduce fill-limited drawings when a scene has backfacing polygons. For example, if you are drawing a sphere, then at any given time, half of its polygons are backfacing. Backface and frontface removal is done after lighting and transformation calculations and before pixel operations. This means that backfacing removal may make transform-limited polygons somewhat slower, but is significantly faster for backfacing fill-limited polygons. You can turn on backfacing removal when you are drawing an object with many backfacing polygons, then turn it off again when the drawing is completed. Backfacing removal causes a noticeable performance drop for transform-limited polygons with five or more sides.

Backfacing removal is especially important on the Personal IRIS and Turbo-Personal IRIS which tend to become fill-limited.

Using Expensive Pixel Modes Efficiently

As with transform modes, expensive pixel modes should be used with care. Pixel operations, in order of increasing cost, with flat-shading being the lowest and texturing the highest, are:

1. flat-shading
2. gouraud shading
3. *z*-buffering
4. alpha-blending
5. texturing

Each of these can independently slow down the pixel fill rate of a polygon. Use *z*-buffering to eliminate the more expensive fill operations of hidden polygons.

Any fill operation can become fill-limiting for large polygons. Clever structuring of drawing can eliminate the need for certain fill operations. For example, if large backgrounds are drawn first, they do not need to be *z*-buffered. It is better to turn *z*-buffering off for the backgrounds and then turn it on again for other objects where it is needed. For example, the *flight* demo uses this technique. The sky and ground are drawn with *z*-buffering turned off, then the polygons lying flat on the ground, *runway* and *grid*, are drawn without suffering a performance penalty. Finally, *z*-buffering is turned on for drawing the mountains and planes.

Alphablending is a significantly expensive pixel operation. A common use of alphablending is with transparency, where the alpha value denotes the opacity of the object. For fully opaque objects, alphablending should be turned off.

The rule is to turn off pixel operations for objects that do not require them and to structure the drawing to minimize their use, without causing excessive toggling of modes.

Balancing Polygon Size and Pixel Operations

To keep the pipeline in balance, the ratio between per pixel work and per vertex work needs to be kept small. For example, with complex lighting transformations, there is more time to fill pixels at the end of the pipeline. With simple vertex operations, there is less time to fill pixels at the end of the pipeline. The pipeline is generally optimized for polygons that are 10 pixel s on a side. If the polygons are too large for the fill-rate to keep up with the rest of the pipeline, the application is fill-rate bound. Smaller polygons balance the pipeline and increase the polygon rate.

Conversely, if the polygons are too small for the rest of the pipeline to keep up with filling, then the application will be transform bound. Larger and fewer polygons, or fewer vertices, balance the pipeline and increase the polygon rate. The simplest possible fill algorithms should be used for drawing very large polygons such as backgrounds.

Clearing the Bitplanes and z-buffer Simultaneously

The most basic per-frame operations are clearing the color and the *z* bitplanes. On some machines, there are optimizations for common, special, cases of these operations. For example, on a system with GT graphics, if you are clearing the screen to black, call **czclear(0,0)**. Call **czclear(0, 0x7fffff)** if you are using a Personal IRIS.

With GT graphics, using **czclear()** is only a performance advantage if the color and *z* arguments are set to the same value, preferably a constant and preferably 0. When you clear the *z* bitplanes to 0, use **lsetdepth()** and an appropriate *z*-compare function to map your *z* values into a range such that the far plane is set to 0. For example:

```
lsetdepth(0x7fffff, 0);
zfunction(ZF_GEQUAL);
czclear(0,0);
```

On a Personal IRIS, using **czclear()** can be up to four times faster than using **clear()** and **zclear()** as long as the z value is equal to **getgdesc(GD_ZMIN)** or **getgdesc(GD_ZMAX)**. Again, appropriate **lsetdepth()** and **zfunction()** settings are needed.

On a VGX system, screen clears are extremely fast. There is no penalty for using **czclear()**, but the color and *z* bitplanes are always cleared sequentially, regardless of the color and *z* values.

See the *Graphics Library Programming Guide* for an in-depth discussion of these topics.

Review of Tuning Methodology

The bottleneck-tuning techniques presented so far produce substantial performance improvements in a short amount of time. A quick application of these techniques may yield performance that is acceptable, so that further tuning is not necessary. However, you may be interested in achieving performance that is demonstrably close to the best the hardware can achieve. To do this, you need to apply a rigorous and systematic analysis.

A detailed analysis involves examining what your program is asking the system to do and then calculating how long it should take, based on the known performance characteristics of the hardware. Compare this calculation of expected performance with the performance actually observed and continue to apply the tuning techniques until the two match more closely. At this point, you have a detailed accounting of how your program spends its time, and you will be in a very strong position both to tune further and to make wise speed vs. quality trade-off decisions.

The following parameters determine performance of most applications:

- Total number of polygons in a frame
- Transform rate for the given polygon type and mode settings
- Number of pixels filled
- Fill rate for the given mode settings
- Time of screen and/or z clear and swapbuffers
- Time of application overhead

To calculate expected performance for a particular code fragment:

1. Determine how many polygons are being drawn and estimate how many pixels they cover on the screen. Have your program count the polygons when you read in the database.

   To determine the number of pixels filled, start by making a visual estimate. Be sure to include surfaces that are hidden behind other surfaces, and notice whether or not backface elimination is enabled. For greater accuracy, use feedback mode and calculate the actual number of pixels filled.

2. Determine the transform and fill rates on the destination hardware platform for the mode settings you are using.

   Refer to the product literature for the destination system to determine some transform and fill rates. Determine others by writing and running small benchmark loops.

3. Divide the number of polygons drawn by the transform rate to get the time spent on per-polygon operations.

4. Divide the number of pixels filled by the fill rate to get the time spent on per-pixel operations.

5. Measure the time spent executing instructions on the CPU.

   To determine time spent executing instructions in the CPU, perform the graphics-stubbing experiment described in "Isolating Bottlenecks in the Pipeline" on page 107.

6. The largest of the three times calculated in steps 3, 4, and 5 is the expected overall performance.

This process takes some effort to complete. In practice, it's best to make a quick start, by making some assumptions, then refine your understanding as you tune and experiment.

Sample Analysis

This section presents a detailed analysis applied to a program that displays a rotating world globe on an IRIS 4D/85GT. The program draws lighted, smooth-shaded, *z*-buffered triangle meshes, and calls **swapbuffers()** and **czclear(0,0)** every frame.

There are approximately 10,000 triangles per frame and most of the polygons average 10 pixels on a side, for a total of 500,000 pixels filled per frame.

On a GT, using the simple lighting model, tmeshes this size have a transform rate of approximately 130K polygons/sec. and a fill rate of 40 Mpix/sec:

```
(10K polys) / (130K polys/sec.) = 77 msec.transform time
(500K pixels) / (40M pixels/sec.) = 12.5 msec.fill time
```

Notice that expected transform time is much greater than the expected fill time, so per-pixel operations will never be the bottleneck and needn't be considered further. Adding the following:

```
8 msec. for a full-screen czclear(0, 0)
```

gives the following total:

```
77msec. + 8 msec. = 85msecs. total time to render a frame
(85 msec. / 16.7 msec.) = 5.089 screen refresh times
```

Notice that this is just over the time for five screen refresh times, so, after synchronization to the screen refresh rate, the expected frame rate is:

```
expected frame rate = 60 Hz/6 = 10 frames/sec.
```

The actual frame rate observed for this application is 3 frames/sec. This means that either the assumptions are incorrect or there is a severe CPU bottleneck.

There did turn out to be inefficiencies in the rendering loop of the application code and after tuning it as described in "Tuning Immediate Mode Drawing" on page 116 the expected rate of 10 frames per second was achieved.

Applying the tuning practices further yields even better performance. Because the calculations indicate that the actual rendering time is just over 5 screen refresh times, a small improvement can push the program under 5 screen refresh times, which gives a frame rate of 12 frames/sec.

The geometry was reconstructed in such a way that it looked almost the same, but contained slightly fewer polygons and 12 frames/sec. was achieved. To do this, the many-sided polygons were decomposed. Also, to take advantage of the parallelism in the graphics pipe, some pairs of adjacent triangles were combined into quads. This is an application of the first technique described in “Using Peak Performance Primitives for Drawing” on page 136.

Experimenting and Benchmarking

Ultimately, you will need to experiment with different rendering techniques and benchmark your ideas, especially when the unexpected happens.

Verify some of these suggestions presented in this chapter for yourself. Try out some of the techniques on a small program that you understand and witness their effect with benchmarks.

Summary of Tuning Techniques

Figure 4-2 is a flowchart of the recommended tuning process.

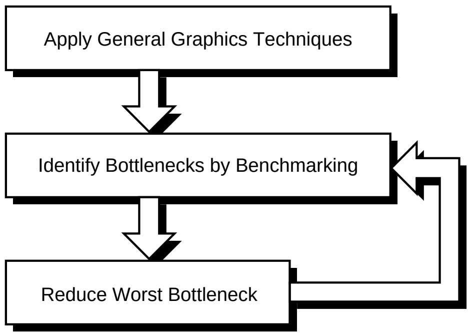

Apply General Graphics Techniques
Identify Bottlenecks by Benchmarking
Reduce Worst Bottleneck

Figure 4-2 Flowchart of the Tuning Process

Overall Graphics Tuning Techniques

These graphics tuning techniques apply to all Silicon Graphics systems. For best results, all the applicable items should be followed.

- Use the high-speed IRIS GL subroutines such as **bgnpolygon()**, **c3f()**, **n3f()**, **cpack()**, **v3f()**, and **endpolygon()**. Do not use the old-style subroutines such as **polf()**, **pmv()**, and **pdr()**.

  For code that does contain old-style commands, call **glcompat(GLC_OLDPOLYGON, FALSE)** in window initialization to turn off IRIS GL outlining of polygons.
- Use three- or four-sided, convex polygons.
- Don't make any graphics calls other than color, normal, texture, and vertex calls in a bgn/end sequence. This includes calls to **lmbind()**, **lmdef()**, **lmcolor()**, and **texbind()**.
- For peak lighting, use a single infinite light with both LOCALVIEWER and TWOSIDE in the lighting model set to 0.0.
- Avoid frequently changing materials and material properties. If material properties must be changed frequently, have a single material and alter just one property. If changing only a couple of material properties, don't use **lmbind()**, use **lmdef()** with a special property array to change specific properties. For example, use:

  ```
  lmcolor(LMC_XXX)
  cpack();
  lmcolor(LMC_COLOR);
  ```

  Use **lmcolor()** if material properties need to be changed per-vertex. Do not change the SHININESS property.
- Use **nmode(NAUTO)**, the default, and pre-normalize all normals.
- When using lighting, do not do any nonuniform scaling in viewing transformations because it forces renormalization of all normals.
- Don't do display-list editing. Use display lists for static objects and do dynamic operations, for example, viewing or matrix operations, in immediate mode.
- For very large display lists, performance may degrade on some machines, although this is going to improve on future machines.

- Group as many line segments and points as possible into single bgnline/endline and bgnpoint/endpoint structures.
- Turn off any modes that are not needed, such as texturing, *z*-buffering, and alpha blending. For example, if drawing transparent objects mixed with opaque objects, turn off alphablending for the opaque objects.
- Sort drawing to minimize expensive mode changes such as **texbind()**, **lmbind()**, and **lmdef()**. Avoid *excessive* changing of any modes, such as **zbuffer()** or **blendfunction()**.
- Use the simplest possible database traversal for drawing.
- Avoid clipping whenever possible.
- Keep all window boundaries within screen limits.
- Arrange windows to be unobstructed by other windows.

POWER Series Techniques

Use these techniques for the POWER Series systems (GTX and VGX):

- Align drawing data to quad-word boundaries (addresses divisible by 16).
- Keep static drawing data that is sent by address to commands like **v3f()**, **n3f()**, **c3f()**, and **t2f()**, separate from other program data, such as immediate mode display flags, or pointers to drawing data, and data sent by value to commands like **cpack()**.

VGX Techniques

Use these techniques for VGX systems:

- Call **subpixel(TRUE)** in window initialization.
- Use connected primitives, tmeshes and qstrips, especially for non-textured objects.
- Use the *float* version of the drawing commands **v3f()**, **c3f()**, **n3f()**, and **t2f()**.
- Minimize the number of calls to **texbind()**, which is currently the most expensive mode change on a VGX.

- For primitives that are not textured, be sure to turn off texturing with **texbind(0,0)**. Otherwise, even if no texture coordinates are issued, texturing remains on and a performance drop is incurred.

  Don't overflow hardware texture memory. There are several factors that effect the size of a texture in texture memory on the VGX. First, VGX textures need borders for wrapping, so their size increases by two (32-bit) texels in each dimension. If the texture is MIPmapped, its size increases by 4/3. The total number of texels are then rounded up to the next multiple of 32.

  Table 4-1 shows how many texels fit within a given texture size for MIPmapped and non-MIPmapped textures on 5 and 10 span VGX systems. See Table 5-3 for texture memory capacity on SkyWriter systems.

Table 4-1 VGX Texture Memory

Texture Size	Non-MIPmapped			MIPmapped		
	Total Texels	5-span	10-span	Total Texels	5-span	10-span
16x16	352	186	465	512	128	320
32x32	1,184	55	137	1,664	39	97
64x64	4,384	14	35	6,016	10	25
128x128	16,928	3	7	22,912	2	5
256x256	66,592	1	2	89,472	0	0
512x512	264,224	0	0	353,664	0	0

- Use one or two component textures rather than three and four-component textures.
- A quad is usually much faster than two independent triangles, so whenever possible use quads rather than independent triangles.
- Use **lrectread()**/**lrectwrite()** in conjunction with **pixmode(PM_SIZE, size)** with the *size* equal to 1, 2, 4, 8, 12, 16, 24, or 32.

- For fast immediate-mode meshes on a VGX, use macros for fixed-length strips to minimize a CPU-limitations. For example, have strips of length 2, 4, 8, and 12 drawn with unrolled macros.

```
#define TMESH_2(dataptr) { \
    bgntmesh(); \
        c3f(dataptr); \
        v3f(dataptr+4); \
        c3f(dataptr+8); \
        v3f(dataptr+12); \
        c3f(dataptr+16); \
        v3f(dataptr+20); \
        c3f(dataptr+24); \
        v3f(dataptr+28); \
        dataptr += 32; \
    endtmesh(); \
    }
```

GT/GTX Techniques

A technique that applies to the GT and GTX machines is to use **czclear(0,0)** with **lsetdepth(0,0x7fffff)** and **zfunction(ZF_GEQUAL)** or **zfunction(ZF_GREATER)**.

Personal IRIS Techniques

Use these techniques for Personal IRIS systems:

- Use **czclear()** with a *z-value* of **getgdesc(GD_ZMIN)**, or **getgdesc(GD_ZMAX)**, and set the *zfunction* to ZF_GEQUAL or ZF_GREATER, or to ZF_LEQUAL or ZF_LESS, respectively.
- Use connected primitives, tmeshes and qstrips.
- Use **shademodel(FLAT)** whenever possible, especially for lines.
- Use **sboxf()** in conjunction with **glcompat(GLC_OLDPOLYGON, FALSE)** for drawing large, flat-shaded, non *z*-buffered, screen-aligned rectangles.

- Use **lrectread()**/**lrectwrite()** in lieu of other pixel commands. Also, use pixel sizes of 8, 16, and 32, because other sizes will be unpacked in software.
- Avoid the use of patterning.

IRIS Indigo Techniques

You'll get the best performance from your IRIS Indigo if you use the simplest modes possible and avoid the use of complex features like fog and texture.

Use these techniques for IRIS Indigo Systems:

- Use **mmode(SINGLE)** whenever possible, especially where multi-matrix mode is not required.
- Use **ortho2()** in MSINGLE mode. It is faster than **ortho()** or **perspective()**.
- Use **shademodel(flat)** where shading is not necessary.
- Use tmeshes wherever possible and draw as many triangles as possible per **bgntmesh()** call.
- Use **sboxf()** wherever possible, which is even more efficient than **tmesh()**.
- Use floating point coordinates where possible, except where memory size is critical, to minimize the time used for floating point conversion. If memory paging is occurring, *shorts* may outperform *floats*.
- Use **subpixel(TRUE)**.

  When anti-aliasing is turned off, lines are scan-converted as closed lines, as in **subpixel(FALSE)** mode. Scan conversion is faster when you use **subpixel(TRUE)**.

  In addition, certain CPU calculations can lead to floating point exceptions, such as dividing by zero. These exceptions are handled either by hardware or by software. If a floating point exception requires software handling, the hardware generates an interrupt. Interrupts require more time to service than floating point operations, thus excessive interrupts can degrade graphics performance.

To check for interrupts:

- Use *osview(1)* to count the interrupts.
- Set the TRAP_FPE environment variable to ALL=COUNT, link your program with *-lfpe* and then run your program. See the **HANDLE_SIGFPES(3C)** man page for more details.

If either of these techniques shows an excess of interrupts (more than 5000 per second), calling **subpixel(TRUE)** will typically reduce the number of interrupts to under 1000 per second. A quiet system generates about 200 interrupts (mouse, clock, timer) per second.

- Use **backface(TRUE)** to eliminate invisible polygons, especially in modes that have slow fill rates, such as *z*-buffering, alpha-blending, and texturing.
- Take advantage of special conditions for 2-D drawing.

  The IRIS Indigo has a high-performance method of drawing 2-D points and lines that makes the **v2()** family of vertex subroutines about twice as fast as their 3-D and 4-D counterparts.

  To take advantage of this special mode, these conditions must be met:

  ```
  mmode(MSINGLE);
  shademodel(FLAT);
  subpixel(TRUE);
  ortho2(...) or equivalent projection transformation
  ```

  In addition, special modes such as *z*-buffering, texture mapping, alphablending, depth-cueing, anti-aliasing, **scrbox()**, feedback and picking must be turned off.
- Minimize the use of z-buffering.

  Avoid using *z*-buffering whenever possible, because the fill rates for *z*-buffering are much slower than normal fill rates. In particular, avoid using *z*-buffered lines, as they are especially slow.

The *z*-buffer is allocated out of main memory. The *z*-buffer is **malloc**-ed to the union of the window and the viewport, so to minimize the amount of memory allocated to the *z*-buffer:

  - Use the smallest possible window and viewport.
  - Avoid using **screenspace()**, because it expands the viewport to the entire screen.

- Minimize the use of alpha-blending, which is even slower than *z*-buffering.

Elan Graphics Techniques

Elan Graphics lets you use advanced IRIS GL features that contribute to better rendering quality and realism of a graphics scene. If you have purchased Elan graphics as an upgrade for your Personal IRIS, Personal IRIS Turbo, or IRIS Indigo, you might not be familiar with these IRIS GL subroutines. See the *Graphics Library Programming Guide* for an introduction to these subroutines, and see the IRIS GL man pages for detailed information on the subroutines presented here.

- Elan Graphics supports the following video formats. In certain cases, product options are required to implement a feature.
  - NTSC (unencoded)
  - PAL (unencoded)
  - Genlock
  - Stereo viewer (requires StereoView Option)
  - 72 Hz screen refresh on certain monitors
  - RS-343 monitor

  Refer to your system documentation for a complete list of features.
- Elan Graphics uses a Single Instruction Multiple Data (SIMD) parallel processing architecture that provides exceptional graphics performance. As a consequence of this architecture, in single buffer mode, you might need to issue a **gflush()** after the last of a series of primitives to force the primitives through the pipeline and expedite graphics processing. In double buffer mode, the **swapbuffers()** call automatically flushes the pipeline; therefore it is not necessary to call

gflush(). This is also known as implicit flushing. Be aware that too many flushes, implicit or explicit, can adversely affect performance.

- Elan Graphics provides hardware and software support for an expanded range of IRIS GL commands. Depending on your system configuration (XS, XS24, or Elan), features such as texturing can be performed in software or hardware. In some cases, this might affect your decision about whether or not to use a feature for performance reasons. Your application can use the **getgdesc()** function to determine the hardware features of your system.
- Elan Graphics does not support alpha bitplanes.
- Antialiasing is used to make the straight edges of graphics primitives appear to be smoother. Several features provided by Elan Graphics are important in antialiasing, including blending and subpixel positioning, described later in this section.

  Elan Graphics supports the smooth (antialiased) RGB line primitives of the IRIS GL that are enabled with linesmooth(SML_ON). The **linesmooth()** command can be modified by issuing hints to provide further corrections. Elan Graphics supports endpoint correction for antialiased lines using the modifier **linesmooth(SML_END_CORRECT)**; it does not support the “smoother” sampling algorithm **linesmooth(SML_SMOOTHER)**. Because these modifiers are hints, not directives, you do not get an error if you specify an unsupported mode; the system just ignores the modifier.
- Blending modifies pixel colors to achieve certain effects such as antialiasing, transparency, and image compositing. The IRIS GL command **blendfunction()** specifies blending operations.

  Elan Graphics supports blending in hardware. All primitives can be blended, with the exception of antialiased lines and points, which use the blending hardware to determine pixel coverage. The alpha value is ignored for these primitives. Pixel blends work best in 24-bit single buffered RGB mode. In double buffered RGB mode, the blend quality degrades.

  Because Elan Graphics does not support alpha planes, all blend options that use destination alpha are inoperable.

- Geometry is clipped to a viewing volume before it is rendered. The projection transformation usually defines the six clipping planes of the viewing volume.

  Elan Graphics supports six additional user-defined clipping planes that can be used to cross-section a geometry, or create an asymmetrical viewing volume. Use the **clipplanes()** command to create up to six arbitrarily placed clipping planes.
- The IRIS GL provides colormap commands that let you reorganize the system colormap; however, Elan Graphics does not support this, so the colormap commands **multimap()**, **cyclemap()**, and **setmap()** are not supported.
- Dithering is used to expand the range of colors that can be created from a group of color components and to provide smooth color transitions. Dithering is user-controllable with the **dither()** command.

  Dithering is the default on all Elan Graphics configurations. To improve clearing performance, dithering is disabled when performing a **clear()** or **czclear()** on systems with 24-bit frame buffers. On 8-bitplane systems, dithering is disabled when performing a **clear()** in color index mode.
- Fog is used to simulate atmospheric conditions. The IRIS GL provides fog commands to create fog that is calculated per-pixel or per-vertex for fogged primitives.

  Elan Graphics supports **fogvertex()** with the per-vertex fog modes FG_VTX_EXP, FG_VTX_EXP2, and FG_VTX_LIN. Elan Graphics does not support the per-pixel fog modes.
- Elan Graphics provides advanced lighting features:
  – Two-sided lighting—TWOSIDE property
  – Spotlights—SPOTDIRECTION and SPOTLIGHT properties
  – 3 factor attenuation—ATTENUATION and ATTENUATION2 properties
- Elan Graphics supports these **pixmode()** operations:
  – PM_EXPAND—expand pixel data
  – PM_ZDATA—write z information to the z-buffer
  – PM_ADD24—add a value to each pixel (supported in software)

- PM_SHIFT—shift pixel data left or right (supported in software)
- PM_TTOB—make pixel fill direction top-to-bottom (see performance information at the end of this section)
- PM_RTOL—make pixel fill direction right-to-left
- PM_SIZE—specify the number of bits per pixel for pixel packing. Values of 1, 8, 16, and 32 are supported in hardware and give the best performance.

When reading/writing pixels, you get the fastest performance by setting the pixel fill direction to top-to-bottom with **pixmode(PM_TTOB, 1)**. Top-to-bottom is not the default fill direction, so it must be set explicitly. Pixel reads on IRIS Crimson systems may be slightly slower than on Personal IRIS and IRIS Indigo XS, XS24, and Elan systems.

- Elan Graphics makes four bitplanes from the z-buffer available for allocation to stencil operations. When using stencil planes, the resolution of the z-buffer is reduced.
- Elan Graphics uses MicroPixel subpixel positioning to retain vertex positions rather than snapping vertices to pixel centers. The IRIS GL command **subpixel()** was used in earlier systems to enable subpixel positioning.

  With Elan Graphics, for polygons, all drawing is performed with subpixel precision, regardless of the setting of **subpixel()**. For lines and points, the **subpixel()** command is operable. Performance is excellent in both modes, but **subpixel(TRUE)** is recommended for the best quality rendering and for compatibility with high-end IRIS systems.
- All Elan Graphics texture mapping is performed in software. As a result, textured primitives do not run at high performance rates.
- The z-buffer stores depth information for graphics primitives. Elan Graphics supports z-buffer operations on systems that have z-buffer hardware installed (default on Elan, optional on XS and XS24, not available on Entry systems). There is no z-buffer support for systems that do not have hardware z-buffers. The low bit of the 24-bit hardware z-buffer is reserved for fast clears and should be ignored when reading data back from the z-buffer. The remaining 23 bits contain the valid z data.

- Drawing into the z-buffer is enabled with **zdraw()**. **zdraw()** works on Elan graphics hardware except under these conditions:
  - Drawing into a clipped window
  - Using read-modify/write operations, for example, antialiasing or **logicop()**.

Benchmarking Techniques

- Use a clock with an accuracy of at least 1/100th of a second to make timing measurements.
- Always call **finish()** before starting the clock and before checking the clock for elapsed time. This routine returns when the graphics pipe is cleared of commands.
- Take measurements over a period of at least a second while repeating the rendering under test to remove the effect of extra timing and system overhead.
- Benchmark in single buffer mode. Double buffer mode forces frame times to be integer multiples of 1/60th of a second.
- Benchmark static frames to find bottlenecks in a given drawing scenario. Benchmark interactions between frames once the drawing scenario is fixed.

Bottleneck Techniques

Use these techniques to reduce specific bottlenecks. In some cases, they may increase performance in one area at the expense of reduced performance in another area. Therefore, these techniques should not be used until you know where the current bottlenecks are.

Reducing CPU Bottlenecks

Use these techniques when you know that your program is CPU-limited:

- Use single dimensional arrays traversed with a pointer that always holds the address for the current drawing command, rather than array-element addressing or multidimensional array accesses.

  ```
  Bad:  v3f(&data[i][j][k]);
  Good: v3f(dataptr);
  ```

  This sample code is efficient code for outputing a single polygon:

  ```
  bgnpolygon();
      n3f(ptr);
      v3f(ptr+4);
      n3f(ptr+8);
      v3f(ptr+12);
      n3f(ptr+16);
      v3f(ptr+20);
      n3f(ptr+24);
      v3f(ptr+28);
      endpolygon();
      ptr += 32;
  ```

- Keep all static drawing data for a given object together in a single contiguous array traversed with a single pointer. Keep this data separate from other program data, such as pointers to drawing data, or interpreter flags.
- Use *switch* statements over multiple *if-else-if* structures.
- Avoid run-time typecasting of values. Typecasting of pointers is performed at compile-time and is acceptable.
- Prototype frequently called subroutines to eliminate run-time typecasting.
- If the program requires floating point calculations, use the *-float* option on the compile line.

Reducing Polygon Operation Bottlenecks

Use these techniques to reduce bottlenecks with per-polygon operations, such as coordinate transformations, lighting, depth-cueing, clipping and concave decomposition. If these operations cannot be simplified, then techniques to reduce the number of such polygons may be needed. Some such techniques are suggested in items 6-8.

- Use connected primitives and put at least 8 in a sequence, 12 to 16 if possible.
- Turn off **backface()** and/or **frontface()** for severely transform-limited drawing, but be careful not to create a fill-limitation.
- Use the simplest possible lighting model, a single infinite light with an infinite viewer. For some local effects, try substituting local lights with infinite lights that have a local viewer.
- Use **nmode(NAUTO)** and prenormalize all normals.
- Minimize the use of depth-cueing. It is very expensive.
- Do not use nonuniform scaling operations or nonorthogonal viewing matrices when using lighting.
- Use only three- or four-sided convex polygons and use **concave(FALSE)**, the default. If necessary, preprocess data to triangulate 5+ sided polygons and/or concave polygons, or do this as a separate process on a separate processor.
- Use CPU backface elimination on a separate processor for backfacing lines, or to save very complex lighting operations that can't be otherwise avoided.
- Use culling on a separate processor to eliminate objects or polygons that will be out of sight or to small to see.

Reducing Pixel Operation Bottlenecks

Use these techniques when you know that you have a bottleneck in per-pixel operations. This kind of bottleneck can be caused by drawing large polygons and/or by using fill operations such as *z*-buffering, alphablending, or texturing.

- Use **backface()** and/or **frontface()** to eliminate all pixel operations for backfacing polygons. This can improve fill-limited performance by a factor of 2.
- Organize the drawing to minimize fill operations. For example, if the scene has large background polygons, then draw them first non *z*-buffered in front, then render the more complex *z*-buffered objects.
- On a VGX, when texturing geometry, using *z*-buffering improves the fill-rate of polygons that are behind what is currently drawing. However, this may not be true on future hardware.

References

Kane, Gerry, *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ 1989.

Chapter 5

Programming Visual Simulation Applications for SkyWriter Systems

This chapter tells you how to program and tune visual simulation applications for SkyWriter systems. It discusses the special graphics features, multiple pipes, and special video formats that are available on the SkyWriter for visual simulation applications.

Using Special Graphics Features

The SkyWriter graphics subsystem provides functional enhancements over the VGX graphics subsystem in a number of areas. These enhancements are documented in the *Graphics Library Programming Guide* and in the online Graphics Library reference (man) pages. To help you port your existing VGX applications quickly, a summary of these changes follows.

Perspective-Correct Texture

The SkyWriter graphics subsystem performs texture perspective correction on a per-pixel basis, so there is no need to use screen subdivision. Thus, the call to **scrsubdivide()** should not be made when running on a SkyWriter system.

Perspective Correct Fog

On a VGX, the fog calculation is performed on a per-vertex basis. The resultant fog color at each vertex is then Gouraud-interpolated across the primitive. SkyWriter has the ability to perform fog calculations on a per-pixel basis. Per-pixel perspective correction is performed for the fog calculation just as it is for the texture calculation. As a result, per-pixel fog is more accurate than per-vertex fog. Furthermore, SkyWriter can usually perform per-pixel fog calculations faster than a VGX can perform per-vertex fog calculations. For a detailed discussion of fog, including equations, see Volume II of the *Graphics Library Programming Guide*.

To take advantage of per-pixel fog, the arguments to **fogvertex()** must be changed. Originally, on VGX systems, it was possible to specify only an exponential fog function. Now, both linear and exponential fog functions can be specified for both VGX and SkyWriter systems. The FG_DEFINE parameter to **fogvertex()** is obsolete and is replaced with one of the following modes:

```
FG_VTX_EXP2
FG_VTX_EXP
FG_VTX_LIN
FG_PIX_EXP2
FG_PIX_EXP
FG_PIX_LIN
```

On a VGX, to specify per-vertex exponential or exponential-squared fog calculations, replace FG_DEFINE with FG_VTX_EXP or FG_VTX_EXP2. Likewise to specify per-vertex linear fog use FG_VTX_LIN.

On a SkyWriter, you should use per-pixel fog calculations. Use FG_PIX_EXP or FG_PIX_EXP2 to specify per-pixel exponential or exponential-squared fog. Similarly, use FG_PIX_LIN for per-pixel linear fog. Notice that the necessary parameters for linear fog are different from those for exponential fog. To maximize the accuracy of the per-pixel fog calculation, minimize the ratio of the distance from the eye to the far clipping plane over the distance from the eye to the near clipping plane. Also, maximize the **lsetdepth()** range. See the *Graphics Library Programming Guide* for instructions on setting up these conditions.

Trilinear MIPmap Filter

SkyWriter supports a trilinear filter operation when MIPmapping (VGX allows up to bilinear). See the *Graphics Library Programming Guide* for a description of texture filters. Using a trilinear filter improves the texture quality at the expense of texture fill performance. To use trilinear filtering, call **texdef2d()** with a minification filter (TX_MINFILTER) of TX_MIPMAP_TRILINEAR.

A fill rate summary is included in “Performance” on page 181. The SkyWriter fill performance, using a trilinear minification filter, is better than the VGX fill performance using a bilinear minification filter.

Outlining Polygons With Antialiased Lines

Another significant new feature for SkyWriter is the ability to antialias polygon edges without requiring a preliminary sort of the database. The basic idea is to first draw the scene as filled polygons and then redraw the portion of the scene that is to be antialiased using **polymode(PYM_LINE_FAST)**, which draws antialiased edges around the polygons. When the scene is redrawn, the antialiased polygon edges will be properly blended into the background. The following sample code fragment illustrates the use of this technique:

```
/* draw lines antialiased */
linesmooth(SML_ON | SML_SMOOTHER);

while (1) {
    /* Setup to draw the scene as polygons */
    polymode(PYM_FILL);
    zfunction(ZF_LEQUAL);
    blendfunction(BF_ONE,BF_ZERO);

    /* draw the scene */
    draw_scene_routine();

    /* Setup to redraw antialiased edges */
    zfunction(ZF_LESS);
    blendfunction(BF_SA,BF_MSA);
    polymode(PYM_LINE_FAST);

    /* draw portions of the scene to be antialiased */
    draw_portion_of_scene_to_be_antialiased_routine();
}
```

SkyWriter decomposes all polygons into triangles before rendering. Using **polymode(PYM_LINE_FAST)** draws antialiased edges after the polygon has been decomposed into triangles. As a result, if you use this polymode to antialias polygons with four or more edges, you may notice some minor visual differences on the interior of these polygons. Because drawing antialiased edges takes time, do so only where necessary.

Multiple Graphics Pipelines

The SkyWriter system contains two complete hardware graphics pipelines that drive two independent video output channels. On a system with two graphics display monitors, this feature is called *dual-head*. SkyWriter also includes a video multiplexer that can be used to produce a single output channel consisting of interleaved frames from the two pipelines, giving twice the graphics performance of a single pipeline. This feature is called *hyper-pipe*.

Figure 5-1 shows the hyper-pipe configuration.

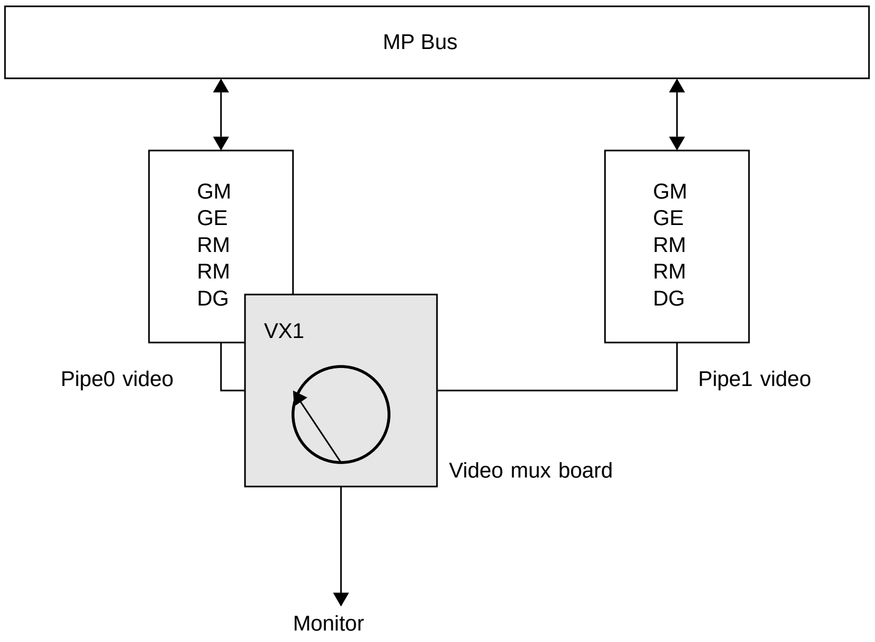


Figure 5-1 Hyper-pipe Configuration

See the *SkyWriter Owner's Guide* for complete instructions on setting up and configuring your SkyWriter to use these modes.

The sections that follow describe the application program interface to control multiple pipes and suggest ways to structure an application to use them effectively.

Pixel-Replicating Video Format

The *pixel-replicating* (PR) video format enables programs to draw into 512x640 pixel region of the frame buffer and fill the drawing area of a 1024x1280 monitor or projector. Video hardware performs 2x2 pixel replication, so performance is identical to that obtained by drawing into a 512x640-pixel window with the default 1024x1280 video format.

To use PR, you must code your application to render into the lower left quarter of the screen and put the display system into PR mode, using the *pr60* argument to the *setmon* command. See *setmon(1g)* for instructions on changing video modes.

Using Multiple Pipes

This section describes what application programs must do to use multiple graphics pipelines, both as independent channels and in hyper-pipe mode.

Window Manager Access to Multiple Pipes

During an interactive session with the window manager, you can use the DISPLAY environment variable to control the pipe on which newly-started graphics programs are run. When DISPLAY is set to “unix:0.0”, programs you start will run on pipe 0; when it is set to “unix:0.1”, programs you start will run on pipe 1.

For convenience, the default startup files (*.login*, *.profile*) for *root* and *guest* shells set DISPLAY to a reasonable initial value, if it is not already set. Each screen has a toolchest that can be used to invoke graphics programs. Each toolchest has the DISPLAY variable in its environment set to the correct value for the screen on which it appears, so any application you invoke from a toolchest will inherit this DISPLAY value, and thus appear on the same screen as the toolchest from which it was invoked.

Figure 5-2 shows a diagram of window manager access to multiple pipes. Once a program has begun execution, it is impossible to move it from one pipe to another from the window manager.

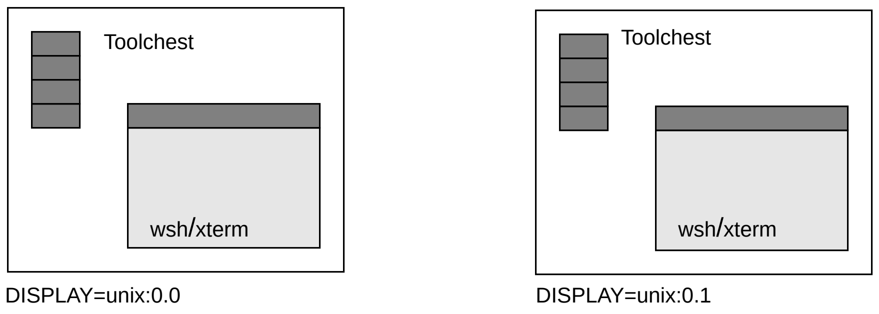


Figure 5-2 Window Manager Access to Multiple Pipes

Program Access to Multiple Pipes

When a program opens a window, it can specify the pipe on which the window is to run. Once a window has been opened, it cannot be moved from one pipe to another. However, a program may achieve the appearance of moving a window from one pipe to another by closing the original window and opening a new window on a new pipe.

There are three different ways in which programs can control the pipe on which windows are opened:

1. Call **winopen()**. The window is opened on the pipe that is specified by the DISPLAY environment variable.

   ```
   # setenv DISPLAY unix:0.1
   # glprogram
   ```

2. Call **scrnselect()** before calling **winopen()**. **scrnselect()** specifies a screen number relative to the current server on which to open a window.

   ```
   main()
   {
       scrnselect(1);
       winopen("my program");
   }
   ```

3. Call **dglopen()** instead of **winopen()**. **dglopen()** specifies which machine, which server, and which screen to use for graphics imaging, in the same format as is used for the DISPLAY variable.

```
main()
{
    dglopen("unix:0.1", DGLLOCAL);
    winopen("my program");
}
```

Hyper-pipe Applications

To operate in hyper-pipe mode, a program must first determine whether it is running on hardware that is capable of hyper-pipe operation. Use **getgdesc(GD_MUXPIPES)** to return the number of *other* graphics pipes able to be video multiplexed with the current graphics pipe. On a SkyWriter, the return value is 1.

The **mswapbuffers()** call has a flag for multiple pipe rendering called DUALDRAW. Both of the display processes must use this flag to indicate that they intend to operate as two cooperative hyper-pipe processes. Each time a process calls **mswapbuffers()**, the video multiplexer is toggled to display the output from that process' pipe. The DUALDRAW flag is used in addition to other **mswapbuffers()** flags. Usually, programs call **mswapbuffers(DUALDRAW | NORMALDRAW)**.

Set up hyper-pipe according to the following rules:

- The process that renders the first, third, fifth, etc. frames must be started on pipe 0, and the process that renders the second, fourth, etc. frames must be started on pipe 1. These processes may be started in either order. The system swaps displayed pipes and blocks processes appropriately, so that frames are displayed in the right order.
- At most, only one hyper-pipe process may render into a given pipe.
- IRIS GL popup menus can be used if they are rendered by the display process that last did **mswapbuffers()**, since its pipe is driving the video signal to the monitor.
- One pipe must be genlocked to the other, using the proper cabling and the *setmon(1g)* command. See the *SkyWriter Owner's Guide* and the *SkyWriter Installation Guide* for information on cabling and genlocking.

Using Pixel Replication with Multiple Pipes

To use PR video format with multiple pipes, issue a *setmon pr60* command for each pipe. In hyper-pipe mode, you must also make sure that one pipe is genlocked to the other. For example, to put the video subsystem into hyper-pipe mode with pixel replication, enter the following commands:

```
setenv DISPLAY :0.1
/usr/gfx/setmon pr60
setenv DISPLAY :0.0
/usr/gfx/setmon -g -t pr60
/usr/gfx/stopgfx
/usr/gfx/startgfx
```

To run the *skyfly* demo in hyper-pipe mode, enter:

```
skyfly -c -o0,0 -w640,512
```

To restore the video subsystem to its default operation (60 Hz), enter:

```
setenv DISPLAY :0.1
/usr/gfx/setmon 60
setenv DISPLAY :0.0
/usr/gfx/setmon 60
/usr/gfx/stopgfx
/usr/gfx/startgfx
```

Process Management

Because SkyWriter is a dual-pipe system, applications are typically made parallel across multiple processes and processors, so that rendering to the two pipes occurs in parallel.

This section discusses the IRIX mechanisms for parallel programming, including process decomposition, synchronization, and shared memory. In addition, this section outlines example programming models for dual-channel and hyper-pipe visual simulation applications.

See *Parallel Programming on Silicon Graphics Computer Systems* for more information on parallel programming. See also Appendix E, “Using Graphics and Share Groups,” in the *Graphics Library Programming Guide*, for additional information on using graphics in shared processes.

Programming Model

A basic process/memory configuration for a dual-pipe visual simulation application is diagrammed in Figure 5-3. This is not the only model possible—it is just one example.

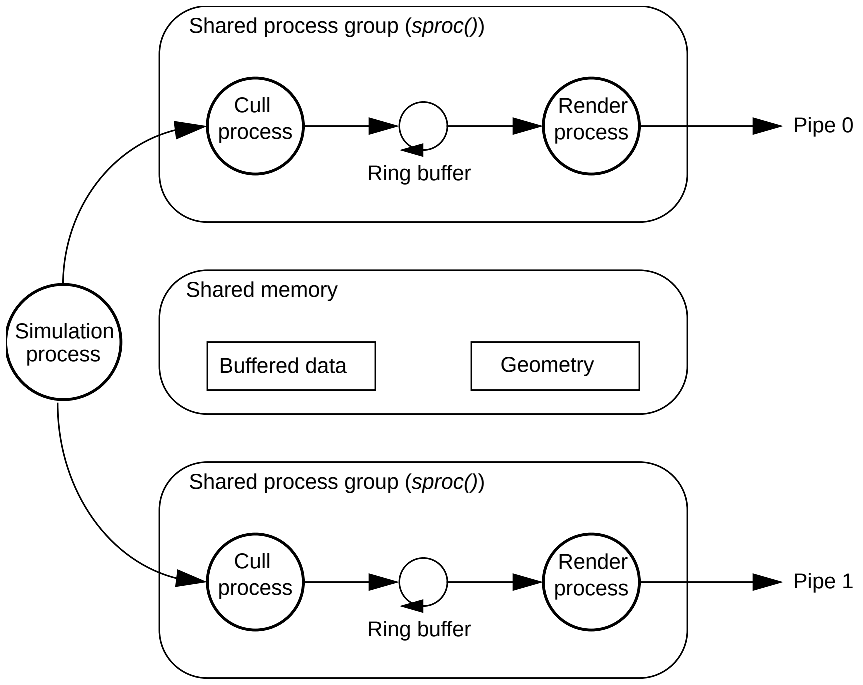


Figure 5-3 Model of Example Shared Process/Memory Configuration

In this example, the visual simulation problem is decomposed into processes of three types: *simulation*, *cull*, and *render*. The simulation process controls the motion of the viewer through the database, computes flight dynamics, object intersections, and so on. In simple cases, a single process can perform all of these functions. In more complex cases, this process can receive information from processes running on other CPUs within the same system or on other systems attached to a network.

The cull process generates a display list of things that are in the viewing frustum and are thus potentially visible. It may also perform scene management functions, such as level-of-detail selection. The display that is generated by the cull process is then rendered by the render process.

In the model shown in Figure 5-3, there are two cull/render process pairs, one for each pipe, and a single simulation process which controls these two “software pipes.” The synchronization of the processes differs for the single-channel hyper-pipe mode and the dual-channel mode.

Hyper-pipe Mode

Figure 5-4 shows a timing diagram for hyper-pipe mode.

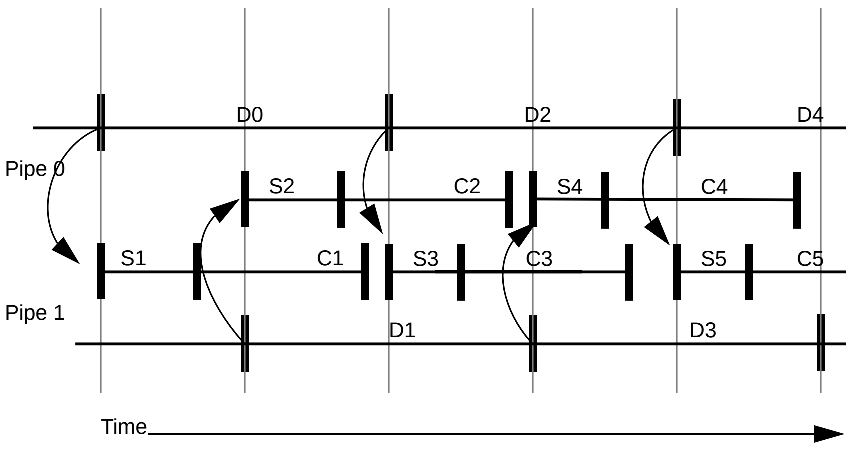


Figure 5-4 Hyper-pipe Mode Timing

In hyper-pipe mode, the drawing processes are automatically synchronized by **mswapbuffers()** as described in “Hyper-pipe Applications” on page 171. They are staggered so that the combined frame rate is exactly twice that of a single pipe. To take advantage of this, the simulation process is triggered by the end of the draw process so that the simulation can sample inputs with regular frequency, that is, at the combined frame rate. Once the simulation has finished, it triggers the cull process of the current pipe.

The cull process places pointers to graphical objects into a ring buffer, and the render process renders the specified graphical objects. Thus, the cull and render processes operate on the same frame in parallel. Once the render process finishes, it triggers the simulation process to work on a future frame for the other pipe.

Dual-channel Mode

Dual-channel operation is typically used to display a different out-the-window view on each pipe of a dual-head system. In this case, the frames on each pipe should be drawn at the same time and not staggered as in the hyper-pipe case. This requires a different synchronization of processes, as shown in Figure 5-5.

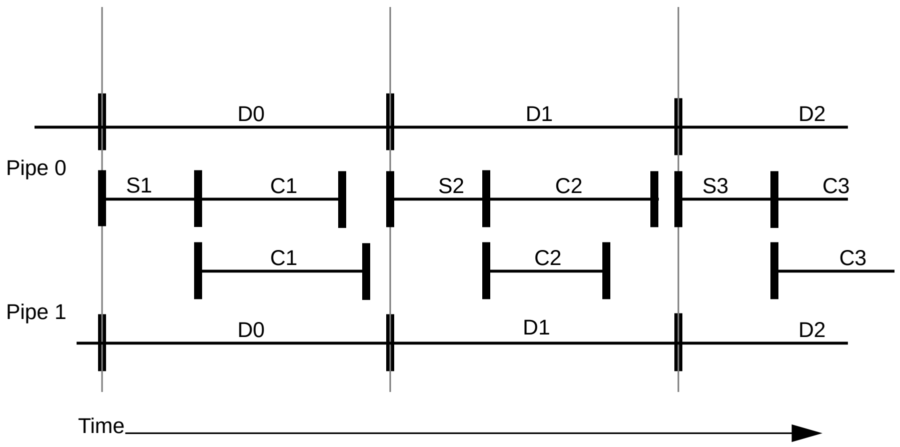


Figure 5-5 Dual-Channel Mode Timing

In Figure 5-5, the simulation process triggers both culls, which work on different views. The simulation then waits for both render processes to finish and then starts on the next frame. In this case it is important that the ring buffer be large enough to contain two complete frames.

In both examples, the simulation and cull processes can be working on different frames at the same time. Because of this, the data that is modified by the simulation and passed to the cull must be buffered so that the proper data is used by the proper frame. Because there are processes working on a maximum of two different frames at the same time, it is necessary to double-buffer only the variable data.

One issue related to parallelism is *latency* (transport delay), the time elapsed from sampling inputs to viewing the resulting output. In hyper-pipe mode, the frame rate is, ideally, doubled over that of a single pipe, but the latency is not halved because the doubling of the frame rate is achieved through parallelism. Latency is increased by the number of stages in the rendering pipeline and is dependent on process synchronization.

Typically, although there are two independent software pipes, composed of the cull/render pairs, there is only one database. Instead of wasting memory by giving each software pipe its own copy of the database, use shared memory to store the database, so that both pipes can access it. Shared memory is also used to pass the buffered viewer and object positions from the simulation to the cull process.

IRIX mechanisms for process management, process synchronization, and shared memory are discussed next.

IRIX Support for Parallel Programming

The type of parallel programming most useful to visual simulation applications is coarse-grained, heterogeneous parallelism, in which processes do large chunks of different kinds of work. The example programming model uses this kind of parallelism in two ways. It processes information for two hardware pipes in parallel and it performs the simulation, culling, and rendering for each pipe in parallel.

The IRIX mechanisms to achieve coarse-grained, heterogeneous parallelism are **sproc()** and **fork()**. Both of these calls clone another identical process, but forked processes acquire their own virtual address space, while **sproc**'ed processes form a *share group* which shares the same virtual address space. Since the Graphics Library's internal data structures are not multi-buffered, two or more rendering processes cannot share the same address space—they *must* be forked.

Forked processes do not share the same virtual address space, but they can obtain pieces of shareable memory from the operating system. The IRIX **mmap()** call is used to map a file into the address space of a process. When multiple processes map the same file, they share it, providing for communication and shared data. Memory can be allocated out of a mapped file using **acreate()** to create a shared memory arena and **amalloc()/afree()** to manipulate memory in the arena.

Process synchronization is achieved through the use of *semaphores*. Semaphores are used to trigger processes and to control access to shared data. **usinit()** creates a shared memory area in which semaphores are created. **uspsema()** and **usvsema()** implement the classic **P()** and **V()** semaphore operations.

In some cases it may be useful to control which processes run on which processors explicitly. For example, lock a critical process, such as render, to a single processor so that it may not be preempted by another process. The exact process/processor mapping depends on the number of processes and their relative priority, as well as the number of processors. The system commands for multiprocessing control are the *mpadmin* program, and the **sysmp()** system call. Both commands provide for some form of process restriction and processor isolation. However, you should be careful to avoid starvation and deadlock problems when using them.

Sample Code

Sample source code which implements both the hyper-pipe and dual-channel programming models may be found in */usr/people/4Dgifts/examples/skywriter*. The program is called *skyfly* and may be run from *buttonfly* or compiled in the 4Dgifts directory.

Mouse Input

Single-process IRIS GL programs can call IRIS GL input routines, for example, **qread()** and **getvaluator()**, to get mouse input relative to the windows they open. However, in complex, multi-process applications, the process that opens and renders to a given window is not necessarily the process that must receive input relative to that window. To handle cases like this, X Window System input commands must be used.

Part of the *skyfly* sample code includes an input system implemented entirely with *Xlib* calls, which closely mimic IRIS GL input calls. This code can be found in the file *xinput.c*.

The initialization routine **openXinput()** creates an invisible window over the whole screen. This window allows the input code to trap all the input events as they are sent from the X server. A program using this input scheme should call **openXinput()** during its initialization sequence. The routine **closeXinput()** should be called during the program's shutdown sequence to restore states modified by **openXinput()**.

The interface routines **Xgetbutton()** and **Xgetvaluator()** behave exactly as their IRIS GL relatives **getbutton()** and **getvaluator()—** they return mouse *x* and *y* coordinate information.

The event handling routine **flushQueue()** is invoked internally by **Xgetbutton()** or **Xgetvaluator()** when they determine that there are X input events waiting to be processed. It is not necessary to call **flushQueue()** from the application. The tasks accomplished by **flushQueue()** include processing all the queued mouse and button events and translating their meanings from the X to the IRIS GL specification.

In summary, you can call the sample input routines from *skyfly/xinput.c* as follows:

- Call **openXinput()** during the application's initialization phase.
- Call **Xgetbutton()** and **Xgetvaluator()** instead of **getbutton()** and **getvaluator()**.
- Call **closeXinput()** during the application's shutdown phase.

Guidelines for Visual Simulation Applications

The following section is a collection of hints and rules-of-thumb that may help you design your visual simulation application.

How to Use Texturing

The large range of texturing options in the SkyWriter system allow great flexibility, but they can be confusing. The following suggestions are for texturing in visual simulation applications, to increase realism while minimizing the impact on performance.

A texture is classified by the number of 8-bit components it uses and the filters used for minification and magnification. A texture can have from one to four components. The one- and two-component versions are for intensity maps, with or without alpha, and the three- and four-component versions are for color maps, with and without alpha. The possible filters are point-sampling (no filter), bilinear, MIPmapped point, MIPmapped-linear, MIPmapped bilinear and MIPmapped trilinear. The general rule-of-thumb is the more components, and the more complicated the filter, the lower the fill rate. A table of fill rates for the various texturing modes is included in "Performance" on page 181.

Choosing Texture Options

In visual simulation, fill rate performance is a prime concern. For this reason most texturing is one component. The complex filters have a smaller performance penalty on a Sky Writer than on a VGX, but should still be used cautiously. Fortunately, a one-component texture is almost always sufficient, if used wisely.

Using blending texture environments, you can use a one-component intensity map to simulate grass, soil, snow, water, sky, etc., merely by using a different environment blend color. If this is not sufficient, you can modulate the color of the base geometry to change color on a per-polygon basis.

For texture maps that contain high-frequency information, MIPmapping minification is required, to avoid the scintillation on distant textures caused by aliasing. Bilinear interpolated magnification is also required, to smooth out under-sampling when the texture is magnified in the foreground. For lower frequency maps, point sampling may be adequate, and it will fill faster when 3 or 4 components are being used.

Selecting or generating an interesting intensity map is vital to creating a convincing simulation, so do not neglect this stage of development.

Managing Texture Memory

Within the graphics subsystem, there is a fixed-size, special-purpose memory, where a texture must reside when it is being used for active drawing.

Multiple texturing programs may run simultaneously, and each may define and use multiple textures, so the texture memory is managed as a virtual memory. The operating system and Graphics Library move textures into and out of texture memory automatically, but the cost of moving textures is high. For this reason, most visual simulation applications should restrict their use of textures to the amount that will fit in texture memory without any swapping.

A detailed description of the amount of texture memory available is included in "Performance" on page 181. Also see "VGX Techniques" on page 151.

Other Texturing Techniques

If possible, turning texturing off for distant objects may provide a significant performance increase. This is because texture considerably decreases transform rates. Disabling texture on distant objects does not significantly degrade realism, because texturing cues are naturally diminished with distance, as the textures lose their high frequency information through minification.

Texturing information can be used as a substitute for geometry when rendering objects such as trees or signs. Instead of creating a complex geometric representation of a tree, you can simply render one or a few triangles with a texture map that is a picture of a tree. When the object is far enough away, it can be drawn as a single polygon that is oriented dynamically to always face the viewer. When it is very close, this dynamic orientation makes the object appear to rotate, so the object is better drawn as two or more intersecting polygons. It will then have the appearance of width when viewed from an angle, but it won't appear to rotate.

How to Use Antialiasing

Aliasing can be a disturbing artifact, especially when using video formats that enlarge the displayed image. Aliasing is a by product of the binary decision as to whether or not to paint a pixel. Sometimes this decision results in over-coverage, and sometimes in under-coverage of a pixel. The IRIS GL provides several options for antialiasing. However, many of these are prohibitively slow (the accumulation buffer) or overly restrictive (**polysmooth()**) for use in visual simulation. The desirable technique is to use

polymode(PYM_LINE_FAST), as described in “Using Special Graphics Features” on page 165.

Aliasing is most disturbing along silhouette edges against backgrounds of a drastically different color. Objects that create such silhouettes are prime candidates for antialiasing. Because antialiasing is a moderately expensive operation, you must decide which objects need antialiasing by playing off the gain in image quality vs. performance degradation. This is a subjective, iterative process.

By determining which polygons in an object statistically contribute to the majority of silhouette edges, you can antialias only these polygons. When rendering during the second pass, render an object that is composed of this selected subset of the geometry of the original object with antialiasing turned on. This technique reduces the transform time associated with the antialiasing stage.

You can further reduce the amount of geometry drawn during the second pass by noticing that abutting polygons share an edge that would be antialiased twice. It may be possible to prune polygons that abut with other polygons on their entire perimeter, since these polygons do not contribute to any unique edges.

Note that the **polymode(PYM_LINE_FAST)** technique only works on the edges of polygons on convex regions of objects. It cannot be used to antialias concave regions of objects, nor lines created by the intersection of self-interesting polygons, nor lines created when an edge of a polygon abuts with the interior of another polygon.

Performance

IRIS GL Tuning Tools and References

For a comprehensive treatment of the theory and practice of graphics tuning, see Chapter 3, “Using GLdebug.” Besides giving techniques for tuning existing programs, Chapter 3 provides detailed advice to help you design new programs, including recommended data structures and traversal methods for fast rendering.

SkyWriter Transform and Fill Rates

Table 5-1 shows the SkyWriter per-pipe transform rates.

Table 5-1 SkyWriter Per-Pipe Transform Rates

Primitive Type	Transform Rate (Primitives per second)
Independent Triangle, Gouraud-Shaded	150K
Triangle Mesh, Textured	75K
Independent Triangle, Fogged	150K
Independent Triangle, Textured, Fogged	75K
Independent Triangle, Textured, Fogged, Antialiased	35K

Table 5-2 shows the SkyWriter per-pipe fill rates.

Table 5-2 SkyWriter Per-Pipe Fill Rates

Polygon Type	Fill Rate
Non-*z*-buffered	193M
z-Buffered	96M
Blended	57M
Textured (Not Fogged / Fogged)	
Point-Sampled, 1- or 2-Component	50M / 40M
Point-Sampled, 3- or 4-Component	50M / 38M
Linear, 1- or 2-Component	50M / 40M
Linear, 3- or 4-Component	50M / 38M
Bilinear, 1- or 2-Component	47-50M / 40M
Bilinear, 3- or 4-Component	33M / 27M
Trilinear, 1- or 2-Component	30-44M / 25-34M
Trilinear, 3- or 4-Component	20M / 17M

You should be aware of several performance issues when porting an application from VGX to SkyWriter. Like VGX, SkyWriter is tuned to give high performance for meshed non-textured primitives, that is, triangle meshes and quadrilateral strips. For textured primitives, SkyWriter is tuned for textured independent triangles. Independent quadrilaterals are almost as fast as independent triangles. However, there is little or no performance increase over independent textured primitives when using meshed textured primitives.

On SkyWriter, the fill rate when performing texture and per-pixel fog calculations is faster than performing texture and per-vertex fog calculations. If texturing is not being used, the per-vertex fog fill rate may be faster for large polygons. The only time that per-pixel fog becomes computationally expensive is when the perspective matrix is modified, the **lsetdepth()** range is changed, or the fog density is modified.

SkyWriter Texture Memory Management

As with VGX, SkyWriter has a fixed amount of real texture memory which is managed by the kernel and the GL. The amount of texture memory on a SkyWriter is 160K texels when the accumulation buffer is not configured. Although each texel can hold from one to four components, the entire texel is used regardless of the number of components in the texture. The memory for non-MIPmapped textures is exactly equal to their area. For example, a 64 by 64 texture uses 4,096 texels. MIPmapped textures on the other hand use 4/3 the area of a top level texture map. For example, a 64 X 64 MIPmapped texture uses 5461 texels. The texture manager manages texture memory using a 32 texel block size. Therefore, round up to the nearest block when computing the amount memory required for a texture. When performing per-pixel fog (see **fogvertex()**), 512 texels of texture memory are reserved for fog calculations.

Binding a texture from the last *n* recently bound textures where the *n* textures fit into texture memory is relatively fast. In general, the speed of a texture bind call depends on how much must be changed to switch to the new texture. Texture binds are fastest when the minification and magnification filters are the same type. For example, use the TX_MIPMAP_BILINEAR minification filter with the TX_BILINEAR magnification filter. The texture bind is almost as fast if the minification filter differs from the magnification filter, but the minification and magnification of the newly bound texture are identical to those of the previously bound texture.

The combination of using texture mapping and the accumulation buffer is not optimized. Thus, calling **acsize()** lengthens the time required to perform subsequent texture binds.

Because texture memory is a shared resource, its availability is affected whenever additional IRIS GL programs that perform texturing are run. The kernel will context switch the texture memory with the rest of an IRIS GL program's state. In general, if multiple texturing applications are running, and there is sufficient texture memory to hold all of the textures for each application, texture maps do not need to be swapped out and reloaded.

Table 5-3 shows how many texels fit within a given texture size for MIPmapped and non-MIPmapped textures on 5 and 10 span SkyWriter systems.

Table 5-3 SkyWriter Texture Memory

Texture Size	Non-MIPmapped			MIPmapped		
	Total Texels	5-span	10-span	Total Texels	5-span	10-span
16x16	256	256	640	352	186	465
32x32	1K	64	160	1,376	47	119
64x64	4K	16	40	6,472	11	29
128x128	16K	4	10	21,856	2	7
256x256	64K	1	2	87,392	0	1
512x512	256K	0	0	349,534	0	0

Practices to Follow for Maximum Performance

Use these techniques to obtain maximum performance on a SkyWriter:

- Always specify **subpixel(TRUE)**.
- Don’t specify negative texture coordinates.
- Limit the range of texture coordinates to (2048/texture size). For example, with a 256 by 256 texture don’t use an *s,t* coordinate greater than 8.0.
- Avoid using any polymode other than PYM_FILL or PYM_LINE_FAST.
- Avoid tiling—tiled textures are slower than non-tiled textures.
- As with any high performance IRIS GL application, clipping should be minimized.
- Fill rate performance is affected by using multi-bank writes, **logicop()**, and **polysmooth()**

  Note: Because SkyWriter supports antialiasing with **polymode(PYM_LINE_FAST)**, the use of polysmooth is not recommended. ♦

Benchmarking Tools

This appendix contains source code for useful benchmarking subroutines and a sample benchmark.

```
/***
 * file:  bench.h
 ***/
#ifdef __cplusplus
extern "C" {
#endif
#include <unistd.h>
#include <signal.h>
/*
 * bench variable declarations
 */
extern float timestart, timestop;
extern float elapsed_time;
extern int gotsignal;
/*
 * bench macros
 */
#define REPEAT10(x) {x x x x x x x x x x}
#define LOOPBENCH(count, btime, benchcode) { \
int _loopbench_i;                            \
gstartclock();                               \
for (_loopbench_i=count;                     \
     _loopbench_i > 0; _loopbench_i--) {     \
            benchcode                        \
    }                               \
btime = ggetelapsedtime();                   \
}
#define SIGBENCH(benchseconds, btime,        \
                 nitems, benchcode) {        \
nitems = 0;                                  \
gotsignal = 0;                               \
signal(SIGALRM, handlesig);                  \
alarm(((benchseconds +1) /2));               \
do {                                            \
            benchcode                           \
            nitems += 2;                        \
}                                               \
while (!gotsignal);                             \
LOOPBENCH(nitems, btime, benchcode);            \
}
extern void handlesig(int sig, ...);
extern int gstartclock(void);
extern float gstopclock(void);
extern float ggetelapsedtime(void);
extern int SetHighPriority(void);
extern void PrintfBenchResults(float btime, int nitems);
#ifdef __cplusplus
};
#endif

/***
 * file: bench.c
 ***/
#include <stdio.h>
#include <gl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/param.h>
#include <sys/prctl.h>
#include <sys/lock.h>
#include <sys/schedctl.h>
#include "bench.h"
/*
 * global variables
 */
int gotsignal = 0;
/* static variables */
static float timestart, timestop;
static float elapsed_time;
static float framerate, polyrate,
cmdrate, cmdsecs;
static struct tms tbuf;
void handlesig(int sig, ...)
{
gotsignal = 1;
}
```

```
int gstartclock(void)
{
  /* flush and put gfx context in pipe */
finish();
timestart = (times(&tbuf) / 100.0);
return(timestart);
}
float ggetelapsedtime(void)
{
/* flush pipe to make sure drawing is
 * really done
 */
finish();
timestop = (times(&tbuf) / 100.0);
elapsed_time = (timestop - timestart);

return(elapsed_time);
}
float gstopclock(void)
{
/* flush pipe */
finish();
timestop = (times(&tbuf) / 100.0);
elapsed_time = (timestop - timestart);
return(timestop);
}
int SetHighPriority(void)
{
/* lock it all down */
printf( "    Locking down all application pages\n" );
if (plock(PROCLOCK) != 0) {
    perror("*** Could not lock down pages - continuing");
    printf("    Using standard unlocked pages\n" );
}
/* give me good priority */
printf( "    Setting nondegrading highest user priority\n");
if (schedctl(NDPRI, 0, NDPHIMAX) != 0) {
    perror("*** Could not give good priority - continuing");
    printf("    Using standard priority\n" );
}
}
```

```
void PrintfBenchResults(float btime, int nitems)
{
float item_time = btime/(float) nitems;
float item_rate = (float)nitems/btime;
printf("nitems=%d, bench_time=%f secs\n", nitems, btime);
printf("time per item:");
if (item_time <= 0.1) {
    printf("%.4f msecs\n",
    item_time *   1000.0);
} else {
    printf("%f secs\n", item_time);
    printf("items per second:");
    if ( item_rate >= 1000.0 )
        printf("%.2fK\n", (item_rate / 1000.0));
    else if ( item_rate >= 1000000.0 )
        printf("%.2fM\n", (item_rate / 1000000.0));
    else
        printf("%.2f\n", item_rate );
}
}
```

This is a sample benchmark using the above code. It benchmarks 10 independent gouraud shaded quads in a display list. Its output as run on a 4D340/VGX follows.

```
/* file: quad.c */
#include <stdio.h>
#include <gl/gl.h>
#include “bench.h”

/* vertex and color data */
float verts[24][4] = {
{10., 0., 0., 0.}, {10., 10., 0., 0.},
{20., 0., 0., 0.}, {20., 10., 0., 0.},
{30., 0., 0., 0.}, {30., 10., 0., 0.},
{40., 0., 0., 0.}, {40., 10., 0., 0.},
{50., 0., 0., 0.}, {50., 10., 0., 0.},
{60., 0., 0., 0.}, {60., 10., 0., 0.},
{70., 0., 0., 0.}, {70., 10., 0., 0.},
{80., 0., 0., 0.}, {80., 10., 0., 0.},
{90., 0., 0., 0.}, {90., 10., 0., 0.},
{100., 0., 0., 0.}, {100., 10., 0., 0.},
{110., 0., 0., 0.}, {110., 10., 0., 0.},
{120., 0., 0., 0.}, {120., 10., 0., 0.},
};
```

```
float colors[24][4] = {
 {0.0,0.0,0.9,0.9}, {0.0,0.9,0.0,0.9},
 {0.0,0.9,0.9,0.9}, {0.9,0.0,0.0,0.9},
 {0.9,0.0,0.9,0.9}, {0.9,0.9,0.0,0.9},
 {0.9,0.9,0.9,0.9}, {0.9,0.0,0.0,0.9},
 {0.9,0.0,0.9,0.9}, {0.9,0.9,0.0,0.9},
 {0.9,0.9,0.9,0.9}, {0.9,0.0,0.0,0.9},
 {0.0,0.0,0.9,0.9}, {0.0,0.9, 0.0,0.9},
 {0.0,0.9,0.9, .9}, {0.9,0.0,0.0,0.9},
 {0.9,0.0,0.9,0.9}, {0.9,0.9, 0.0,0.9},
 {0.9,0.9,0.9,0.9}, {0.9,0.0,0.0,0.9},
 {0.9,0.0,0.9,0.9}, {0.9,0.9,0.0,0.9},
 {0.9,0.9,0.9,0.9}, {0.9,0.0, 0.0,0.9},
};


initwin()
{
foreground();
prefposition(0, 200, 0, 20);
winopen(“quad bench”);
subpixel(TRUE);
RGBmode();
gconfig();
cpack(0x0);
clear();
}
static Object mkquadobj(void)
{
int j, quadobj;
makeobj(quadobj = genobj());
for (j = 0; j < 22; j+=2) {
    bgnpolygon();
        c3f(colors[j]); v3f(verts[j]);
        c3f(colors[j+1]); v3f(verts[j+1]);
        c3f(colors[j+3]); v3f(verts[j+3]);
        c3f(colors[j+2]); v3f(verts[j+2]);
    endpolygon();
}
closeobj();
return quadobj;
}
```

```
main()
{
/* time for SIGBENCH to run */
int benchSeconds = 2;
/* SIGBENCH will write the elapsed time */
float benchTime =0.;
/* SIGBENCH will write the number of times item drawn */
int numItems=0;
/* obj id for object to bench */
Object obj;
initwin();
obj = mkquadobj();
/* run the benchmark 10 times to get an average result */
REPEAT10(
    SIGBENCH(benchSeconds, benchTime, numItems,
             callobj(obj); )
/* 10 quads in the object */
numItems *= 10;
PrintfBenchResults(benchTime, numItems);
)
}
```

An example of the output of this sample benchmark follows:

```
Output from a 340/VGX: (peak rate is 180k independent
gouraud-shaded quads/sec.)
nitems=356980, bench_time=1.980469 secs
time per item:0.0055 msecs
items per second:180.25K
nitems=359880, bench_time=1.988281 secs
time per item:0.0055 msecs
items per second:181.00K
nitems=358100, bench_time=1.988281 secs
time per item:0.0056 msecs
items per second:180.11K
nitems=357940, bench_time=1.980469 secs
time per item:0.0055 msecs
items per second:180.73K
nitems=358700, bench_time=1.988281 secs
time per item:0.0055 msecs
items per second:180.41K
nitems=358420, bench_time=1.980469 secs
time per item:0.0055 msecs
items per second:180.98K
nitems=361400, bench_time=2.000000 secs
```

```
time per item:0.0055 msecs
items per second:180.70K
nitems=358740, bench_time=1.980469 secs
time per item:0.0055 msecs
items per second:181.14K
nitems=360880, bench_time=2.000000 secs
time per item:0.0055 msecs
items per second:180.44K
nitems=357980, bench_time=1.980469 secs
time per item:0.0055 msecs
items per second:180.76K
```

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1489-030.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389