OpenGL on Silicon Graphics Systems

Early Access Release
November 95

Document Number 007-2392-001

CONTRIBUTORS

Written by Renate Kempf and Jed Hartman
Illustrated by Dany Galgani
Edited by Christina Cary
Production by Kay Maitz
Engineering contributions by Allen Akin, David Blythe, Sharon Rose Clay, Tom Davis, Ziv Gigus, David Gorgen, Paul Hansen, Paul Ho, Simon Hui, George Kyriazis, Mark Kilgard, Ed Immenschuh, Steve Moffitt, Dave Shreiner, Joel Tesler, Bill Torzewski, Bill Wehner, Rolf van Widenfelt, Paula Womack, David Yu, and others.

OpenGL on Silicon Graphics Systems
Document Number 007-2392-001

Contents

List of Examples

List of Figures

List of Tables

About This Guide

OpenGL on Silicon Graphics Systems explains how to use the OpenGL® graphics library on Silicon Graphics® systems. The guide expands on the *OpenGL Programming Guide*, which describes implementation-independent aspects of OpenGL. It discusses these major topics:

- Integrating OpenGL programs with the X Window System™
- Using OpenGL extensions
- Debugging OpenGL programs
- Achieving maximum performance

What This Guide Contains

This guide consists of 14 chapters and 3 appendixes:

- Chapter 1, “OpenGL on Silicon Graphics Systems,” introduces the major issues involved in using OpenGL on Silicon Graphics systems.
- Chapter 2, “Using OpenGL and X: Getting Started,” first provides background information for working with OpenGL and the X Window System. You then learn how to display some OpenGL code in an X window with the help of a simple example program.
- Chapter 3, “OpenGL and X: Examples,” first presents two example programs that illustrate how to create a window using IRIS IM or Xlib. It then explains how to integrate text with your OpenGL program.
- Chapter 4, “OpenGL and X: Advanced Topics,” helps you refine your programs. It discusses how to use overlays and popups. It also provides information about pixmaps, visuals and colormaps, and animation.
- Chapter 5, “Introduction to OpenGL Extensions,” explains what OpenGL extensions are and how to check for OpenGL and GLX extension availability.
- Chapter 6, “Texturing Extensions,” explains how to use the texturing extensions, providing example code as appropriate.

- Chapter 7, “Imaging and Blending Extensions,” explains how to use extensions for color conversion (abgr, color table, color matrix), the convolution extension, the histogram/minmax extension, the packed pixel extension, and four color blending extensions.
- Chapter 8, “Miscellaneous OpenGL Extensions,” explains how to use the polygon offset extension, the vertex array extension, and the multisampling extension.
- Chapter 9, “Extensions to GLX,” explains how to use extensions to GLX, the OpenGL extension to the X Window System.
- Chapter 10, “Debugging OpenGL Programs,” explains how to use the OpenGL debugger (ogldebug) and provides some debugging hints.
- Chapter 11, “Tuning Graphics Applications: Fundamentals,” discusses basic principles of tuning graphics applications: pipeline tuning, tuning animations, optimizing cache and memory use, and benchmarking. You need this information as a background for the chapters that follow.
- Chapter 12, “Tuning the Pipeline,” explains how to tune the different parts of the graphics pipeline for an OpenGL program. Example code fragments illustrate how to write your program for optimum performance.
- Chapter 13, “Tuning Graphics Applications: Examples,” provides a detailed discussion of the tuning process for a small example program. It also provides a code fragment that’s helpful for drawing pixels fast.
- Chapter 14, “System-Specific Tuning,” provides information on tuning some specific Silicon Graphics systems: low-end systems, IMPACT™ systems, and RealityEngine™ systems.
- Appendix A, “OpenGL and IRIS GL,” helps you port your IRIS GL™ program to OpenGL by providing a table that contrasts IRIS GL functions and equivalent OpenGL functionality (including extensions).
- Appendix B, “Benchmarks,” lists a sample benchmarking program.
- Appendix C, “Benchmarking Libraries: libpdb and libisfast,” discusses two libraries you can use for benchmarking drawing operations and maintaining a database of the results.

Note that although this guide contains information useful to developers porting from IRIS GL to OpenGL, the primary source of information for porting is the *OpenGL Porting Guide*, available from Silicon Graphics (or via the IRIS Insight™ viewer online).

What You Should Know Before Reading This Guide

To work successfully with this guide, you should be comfortable programming in ANSI C or C++. You should have a fairly good grasp of graphics programming concepts (terms such as "texture map" and "homogeneous coordinates" aren't explained in this guide), and you should be familiar with the OpenGL graphics library. Some familiarity with the X Window System, and with programming for Silicon Graphics platforms in general, is also helpful. If you're a newcomer to any of these topics, see the references listed under "Background Reading."

Background Reading

The following books provide background and complementary information for this guide. Bibliographical information or the Silicon Graphics document number is provided. Books available in hardcopy and by using the IRIS InSight online viewer are marked with **(I)**:

OpenGL and Associated Tools and Libraries

- Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Menlo Park, CA: Addison-Wesley Publishing Company, 1993. **(I)**
- *OpenGL Porting Guide* (007-1797-020) **(I)**
- *IRIS IM Programming Guide* (007-1472-020)
- *IRIS ViewKit Programmer's Guide* 007-2124-002. **(I)**
- *Developer Magic: RapidApp User's Guide* (007-2590-002) **(I)**
- Wernecke, Josie and the Open Inventor Architecture Group. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*, Release 2. Menlo Park, CA: Addison-Wesley Publishing Company, 1994. **(I)**

X Window System: Xlib, X Toolkit, and OSF/Motif

- O'Reilly X Window System Series, Volumes 1, 2, 4, 5, and 6 (referred to in the text as "O'Reilly" with a volume number):
  - Nye, Adrian. *Volume One: Xlib Programming Manual*. Sebastopol, CA: O'Reilly & Associates, 1991.**(I)**

- *Volume Two. Xlib Reference Manual*. Sebastopol, CA: O'Reilly & Associates.
- Nye, Adrian, and Tim O'Reilly. *Volume Four. X Toolkit Intrinsics Programming Manual*. Sebastopol, CA: O'Reilly & Associates, 1992. **(I)**
- Flanagan, David (ed). *Volume Five. X Toolkit Intrinsics Reference Manual*. Sebastopol, CA: O'Reilly & Associates, 1990.
- Heller, Dan. *Volume Six. Motif Programming Manual*. Sebastopol, CA: O'Reilly & Associates.

• Young, Doug. *Application Programming with Xt: Motif Version*

• Kimball, Paul E. *The X Toolkit Cookbook*. Englewood Cliffs, NJ: Prentice Hall, 1995.

• Open Software Foundation. *OSF/Motif Programmer's Guide, Revision 1.2*. Englewood Cliffs, NJ: Prentice Hall, 1993. **(I)**

• Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.2*. Englewood Cliffs, NJ: Prentice Hall, 1993.**(I)**

• Open Software Foundation. *OSF/Motif User's Guide, Revision 1.2*. Englewood Cliffs, NJ: Prentice Hall, 1993.

• Open Software Foundation. *OSF/Motif Style Guide*. Englewood Cliffs, NJ: Prentice Hall. **(I)**

Other Sources

- Kane, Gerry. *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice Hall. 1989.
- *MIPS Compiling and Performance Tuning Guide.* 008-2479-001. **(I)**.

Conventions Used in This Guide

This section explains the typographical and function-naming conventions used in this guide.

Typographical Conventions

This guide uses the following typographical conventions:

Italics — Filenames, IRIX command names, function parameters, and book titles.

`Fixed-width` Code examples and system output.

`Fixed-width bold`
User input.

Bold Function names, with parentheses following the name—for example **glPolygonMode()**, arguments to command line options.

Note: In the “New Functions” sections in the extensions chapters, regular font is used for function names. This avoids large blocks of boldface text, which would be difficult to read.

Function Naming Conventions

This guide refers to a group of similarly named OpenGL functions by a single name, using an asterisk to indicate all the functions whose names start the same way. For instance, **glVertex*()** refers to all functions whose names begin with “glVertex”: **glVertex2s()**, **glVertex3dv()**, **glVertex4fv()**, and so on.

Naming conventions for X-related functions can be confusing, as they depend largely on capitalization to differentiate between groups of functions. For systems on which both OpenGL and IRIS GL are available, the issue is further complicated by the similarity in function names. Here’s a quick guide to old and new function names:

GLX*() IRIS GL mixed-model support

Glx*() IRIS GL support for IRIS IM

glX*() OpenGL support for X

GLw*() OpenGL support for IRIS IM

Note that the (OpenGL) **glX*()** routines are collectively referred to as “GLX”; that term was previously used to refer to the (IRIS GL) **GLX*()** routines. Note, too, that **GLXgetconfig()** (an IRIS GL mixed-model routine) is not the same function as **glXGetConfig()** (a GLX routine). On systems with both IRIS GL and OpenGL, the command

```
IRIS% man glxgetconfig
```

displays both reference pages, one following the other.

Chapter 1

OpenGL on Silicon Graphics Systems

Silicon Graphics systems allow you to write OpenGL applications that are portable and run well across the Silicon Graphics workstation product line. This chapter introduces the basic issues you need to know about if you want to write an OpenGL application for Silicon Graphics systems. The chapter contains the following topics, which are all discussed in more detail elsewhere in this guide:

Using OpenGL With the X Window System

OpenGL is a window-system-independent graphics library. The platform's window system determines where and how the OpenGL application is displayed and how events (user input or other interruptions) are handled. Currently, OpenGL is available for the X Window System, for OS/2™, for Windows NT™, and for Windows™ 95. If you intend your application to run under several window systems, the application's OpenGL calls can remain unchanged but window system calls are different for each window system.

Note: If you plan to have your application run under different window systems, be sure to keep the windowing code isolated as much as possible to minimize the number of files that must be special for each implementation.

All Silicon Graphics systems use the X Window System. Applications on a Silicon Graphics system rely on Xlib calls to manipulate windows and obtain input. An X-based window manager (usually *4Dwm*) handles iconification, window borders, and overlapping windows. The Indigo Magic™ desktop environment is based on X, as is the Silicon Graphics widget set, IRIS IM. IRIS IM is the Silicon Graphics port of OSF/Motif.™

A full introduction to X is beyond the scope of this guide; for detailed information about X, see the sources listed in "Background Reading" on page xxv.

The GLX Extension to the X Window System

The OpenGL extension to the X Window System (GLX) provides a means of creating an OpenGL context and associating it with a drawable window on a computer that uses the X Window System. GLX is provided by Silicon Graphics and other vendors as an adjunct to OpenGL.

For additional information on using GLX, see "The GLX Extension to X" on page 10. More detailed information is in Appendix D, "OpenGL Extensions to the X Window System" of the *OpenGL Programming Guide.* The *glxintro* reference page also provides a good introduction to the topic.

Libraries, Tools, Toolkits, and Widget Sets

When you prepare a program to run with the X Window System, you can choose the level of complexity and control that suits you best, depending on how much time you have and how much control you need.

This section looks at different tools and libraries for working with OpenGL in an X Window System environment, moving from easy-to-use toolkits and libraries with less control to the Xlib library, which is more primitive but offers more control. Most application developers usually write at a higher level than Xlib, but you may find it helpful to understand the basic facts about the lower levels discussed in this guide.

Note that the different tools are not mutually exclusive: You may design most of the interface with one of the higher-level tools, then use Xlib to fine-tune a specific aspect or add something that's otherwise unavailable. Figure 1-1 illustrates the layering:

- IRIS ViewKit™ and Open Inventor™ are layered on top of IRIS IM, which is on top of Xlib.
- GLX links Xlib and OpenGL.
- Open Inventor uses GLX and OpenGL.

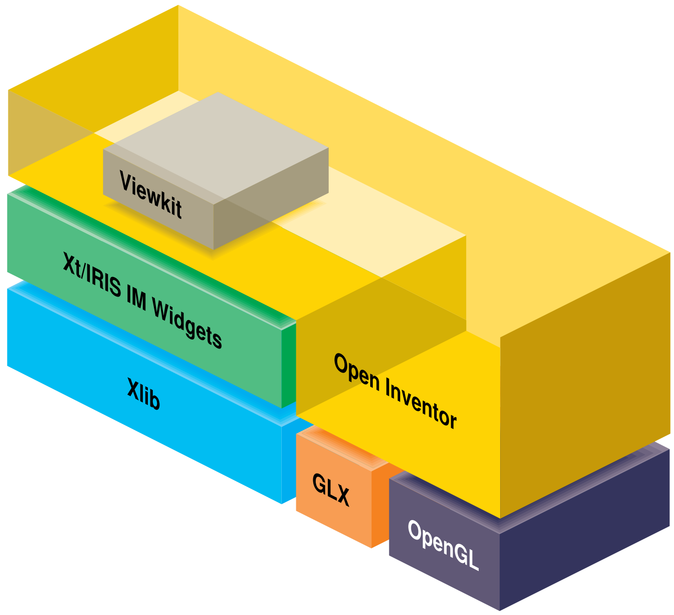


Figure 1-1 How X, OpenGL, and Toolkits are Layered

Note: If you write an application using IRIS Viewkit, OpenInventor, or RapidApp™, the graphical user interface will be visually consistent with the Indigo Magic desktop.

RapidApp

RapidApp is a graphical tool, available from Silicon Graphics, that allows developers to interactively design the user-interface portion of their application. It generates C++ code utilizing IRIS ViewKit (see "IRIS ViewKit" on page 5) for each user-interface component as well as the overall application framework. As with all applications based on ViewKit, IRIS IM (Motif) widgets are the basic building blocks for the user interface. RapidApp is not included in Figure 1-1 because it generates ViewKit and IRIS IM code and is therefore dependent on them in a way different from the rest of the hierarchy.

To speed the development cycle, RapidApp is integrated with a number of the Developer Magic™ tools. This allows developers to quickly design, compile, and test object-oriented applications.

RapidApp also provides easy access to Silicon Graphics specific widgets and components. For instance, you can add an OpenGL widget to a program without having to know much about the underlying details of integrating OpenGL and X.

For more information, see the *Developer Magic: RapidApp User's Guide*, also available online through IRIS InSight.

Open Inventor

The Open Inventor library uses an object-oriented approach to make the creation of interactive 3D graphics applications as easy as possible by letting you use its high-level rendering primitives in a scene graph. It is a useful tool for bypassing the complexity of X and widget sets, as well as many of the complex details of OpenGL.

Open Inventor provides prepackaged tools for viewing, manipulating, and animating 3D objects. It also provides widgets for easy interaction with X and Xt, and a full event-handling system.

In most cases, you use Open Inventor, not the lower-level OpenGL library for rendering from Open Inventor. However, the Open Inventor library provides several widgets for use with X and OpenGL (in subclasses of the SoXtGLWidget class) that you can use if OpenGL rendering is desired. For instance, the SoXtRenderArea widget and its viewer subclasses can all perform OpenGL rendering. SoXtGLWidget is, in turn, a subclass of SoXtComponent, the general Open Inventor class for widgets that perform 3D editing.

Components provide functions to show and hide the associated widgets, set various parameters (such as title and size of the windows), and use callbacks to send data to the calling application. The viewer components based on SoXtRenderArea handle many subsidiary tasks related to viewing 3D objects. Other components handle anything from editing materials and lights in a 3D scene, to copying and pasting 3D objects.

Note that if you're using libInventorXt, you only need to link with libInventorXt (it automatically "exports" all of the routines in libInventor, so you never need to use **-lInventorXt -lInventor**, you only need **-lInventorXt**).

For detailed information on Open Inventor, see *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor,™ Release 2*, published by Addison-Wesley and available online through IRIS InSight.

IRIS ViewKit

The IRIS ViewKit library is a C++ application framework designed to simplify the task of developing applications based on the IRIS IM widget set. The ViewKit framework promotes consistency by providing a common architecture for applications and improves programmer productivity by providing high-level, and in many cases automatic, support for commonly-needed operations.

When you use Viewkit in conjunction with OpenGL, it provides drawing areas that OpenGL can render to.

For more information, see the *IRIS ViewKit Programmer's Guide*, available online through IRIS InSight.

The IRIS IM Widget Set

The IRIS IM widget set is an implementation of OSF/Motif provided by Silicon Graphics. You're strongly encouraged to use IRIS IM when writing software for Silicon Graphics systems. IRIS IM integrates your application with the desktop's interface. If you use it, your application conforms to a consistent look and feel for Silicon Graphics applications. See the sources listed in "Background Reading" on page xxv for further details.

The Xlib Library

The X library, Xlib, provides function calls at a lower level than most application developers want to use. Note that while Xlib offers the greatest amount of control, it also requires that you attend to many details you could otherwise ignore. If you do decide to use Xlib, you are responsible for maintaining the Silicon Graphics user interface standards.

A Note to IRIS GL Users

An application that uses both IRIS GL and X is called a mixed-model program. If you prepared your IRIS GL application to run as a mixed-model program, porting to OpenGL becomes much easier. For porting information, see the *OpenGL Porting Guide*.

Many IRIS GL programs use the built-in windowing interface provided by IRIS GL. In contrast, OpenGL relies on X for all its windowing functionality. If your application uses

IRIS GL functions like **winopen()**, your windowing code needs to be rewritten for X. See the *OpenGL Porting Guide* for more information.

Note that the term "mixed-model program" is no longer relevant when you work with OpenGL, since all OpenGL programs use the native window system for display and event handling. (The OpenGL API, unlike IRIS GL, has no windowing calls).

Extensions to OpenGL

The OpenGL standard is designed to be as portable as possible and also to be expandable with extensions. Extensions may provide new functionality, such as several video extensions, or extend existing functionality, such as blending extensions.

An extension's functions and tokens use a suffix that indicates the availability of that extension:

- EXT is used for extensions reviewed and approved by more than one OpenGL vendor.
- SGI is used for extensions found across the Silicon Graphics product line, although the support for all products may not appear in the same release.
- SGIS is used for extensions found only on a subset of Silicon Graphics platforms.
- SGIX is used for experimental extensions: In future releases, the API for these extensions may change, or they may not be supported at all.

The *glintro* reference page provides a useful introduction to extensions; many extensions are also discussed in detail in the following chapters in this guide:

- Chapter 5, "Introduction to OpenGL Extensions"
- Chapter 6, "Texturing Extensions"
- Chapter 7, "Imaging and Blending Extensions"
- Chapter 8, "Miscellaneous OpenGL Extensions"
- Chapter 9, "Extensions to GLX"

Note that both the X Window System and OpenGL support extensions. GLX is an X extension to support OpenGL. Keep in mind that OpenGL (and GLX) extensions are different from X extensions.

Debugging and Performance Optimization

If you want a fast application, think about performance from the start. While making sure the program runs reliably and bug free is important, it's also essential that you think about performance early on. Applications designed and written without performance considerations can almost never be suitably tuned.

If you want high performance, read the performance chapters in this guide before you start writing the application.

Debugging Your Program

Silicon Graphics provides a variety of debugging tools for use with OpenGL programs:

- The *ogldebug* tool helps you find OpenGL programming errors and discover OpenGL programming style that may slow down your application. You can set breakpoints, step through your program, and collect a variety of information.
- For general-purpose debugging, you can use standard UNIX debugging tools such as *dbx*.
- Also available (for general-purpose debugging) are the CASE tools. For more information on the CASE tools, see *ProDev WorkShop and MegaDev Overview* and *CASEVision/Workshop User's Guide.*

Tuning Your OpenGL Application

The process of tuning graphics applications differs from that of tuning other kinds of applications. This guide provides platform-independent information about tuning your OpenGL application in these chapters:

- Chapter 11, "Tuning Graphics Applications: Fundamentals"
- Chapter 12, "Tuning the Pipeline"
- Chapter 13, "Tuning Graphics Applications: Examples"

In addition, there are tuning issues for particular hardware platforms. They are discussed in Chapter 14, "System-Specific Tuning."

Maximizing Performance With IRIS Performer

The IRIS Performer™ application development environment from Silicon Graphics automatically optimizes graphical applications on the full range of Silicon Graphics systems 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; and state management control to minimize overhead.

Location of Example Source Code

All complete example programs (though not the short code fragments) are available in */usr/share/src/OpenGL* if you have the *ogl_dev.sw.samples* subsystem installed.

Chapter 2

Using OpenGL and X: Getting Started

This chapter first presents background information that you will find useful when working with OpenGL and the X Window System. It then helps you get started right away by discussing a simple example program that displays OpenGL code in an X window. Topics include:

Background and Terminology

To effectively integrate your OpenGL program with the X Window System, you need to understand some basic concepts, discussed in these sections:

- "The X Window System on Silicon Graphics Systems"
- "X Window System Concepts"

Note: If you are unfamiliar with the X Window System, you are urged to learn about it using some of the material listed under "Background Reading" on page xxv.

The X Window System on Silicon Graphics Systems

The X Window System is the only window system provided for Silicon Graphics systems running IRIX 4.0 or later.

X is a network-transparent window system: An application need not be running on the same system on which you view its display. In the X client/server model, you can run programs on the local workstation or remotely on other workstations connected by a network. The X server handles input and output and informs client applications when various events occur. A special client, the window manager, places windows on the screen, handles icons, and manages titles and other window decorations.

When you run an OpenGL program in an X environment, window manipulation and event handling are performed by X functions. Rendering can be done with both X and OpenGL. In general, X is for the user interface and OpenGL is used for rendering 3D scenes or for imaging.

The Silicon Graphics X Server

The X server provided by Silicon Graphics includes some enhancements that not all servers have: Support for visuals with different colormaps, overlay windows, the Display PostScript extension, the Shape extension, the X input extension, the Shared Memory extension, the video control extension, and simultaneous displays on multiple graphics monitors. Specifically for working with OpenGL programs, Silicon Graphics offers the GLX extension discussed in the next section.

To see what extensions to the X Window System are available on your current system, execute *xdpyinfo* and check the extensions named below the “number of extensions” line.

The GLX Extension to X

The GLX extension, which integrates OpenGL and X, is used by X servers that support OpenGL. GLX is both an API and an X extension protocol for supporting OpenGL. GLX routines provide basic interaction between X and OpenGL. Use them, for example, to create a rendering context and bind it to a window.

Compiling With the GLX Extension

To compile a program that uses the GLX extension, include the GLX header file (*/usr/include/GL/glx.h*), which includes relevant X header files and the standard OpenGL header files. If desired, include also the GLU utility library header file (*/usr/include/GL/glu.h*).

Table 2-1 provides an overview of the headers and libraries you need to include.

Table 2-1 Headers and Link Lines for OpenGL and Associated Libraries

Library	Header	Link line
OpenGL	GL/gl.h	-lGL
GLU	GL/glu.h	-lGLU
GLX	GL/glx.h	-lGL (includes GLX and OpenGL)
X11		-lX11

X Window System Concepts

To help you understand how to use your OpenGL program inside the X Window System environment, this section discusses some concepts you will encounter throughout this guide. You learn about

- "Visuals"
- "Drawables"
- "Rendering Contexts"
- "Resources"
- "Colormaps"

Visuals

A standard X visual specifies how the server should map a given pixel value to a color to be displayed on the screen. Different windows on the screen can have different visuals.

Currently, GLX allows RGB rendering to TrueColor and DirectColor visuals and color index rendering to StaticColor or PseudoColor visuals. See Table 4-1 on page 67 for information about the visuals and their supported OpenGL rendering modes. The framebuffer configuration extension allows additional combinations. See "The Framebuffer Configuration Extension" on page 171.

GLX overloads X visuals to include both the standard X definition of a visual and OpenGL specific information about the configuration of the framebuffer and ancillary buffers that might be associated with a drawable. Only those overloaded visuals support both OpenGL and X rendering—GLX therefore requires that an X server support a high minimum baseline of OpenGL functionality.

When you need visual information,

- execute *xdpyinfo* to find out what visuals your system supports
- execute *glxinfo* or *findvis* to find visuals that can be used with OpenGL

  The *findvis* command can actually look for available visuals with certain attributes. See the reference page for more information.

Not all X visuals support OpenGL rendering, but all Silicon Graphics systems provide at least two OpenGL capable visuals. The exact number and type vary among different hardware systems. An RGBA visual is required for any hardware system that supports OpenGL; a color index visual is required only if the hardware requires color index. To determine the OpenGL configuration of a visual, you must use a GLX function.

Visuals are discussed in some detail in "Using Visuals" on page 66. Table 4-1 on page 67 illustrates which X visuals support which type of OpenGL rendering and whether the colormaps for those visuals are writable or not.

Drawables

A drawable is something X can draw into, either a window or a pixmap. A GLX drawable is something both X and OpenGL can draw into, either an OpenGL capable window or a GLX pixmap. (A GLX pixmap is a handle to an X pixmap that is allocated in a special way; see Figure 4-2 on page 77.) Different ways of creating a GLX drawable are discussed in "Drawing-Area Widget Setup and Creation" on page 29, "Creating a Colormap and a Window" on page 42, and "Using Pixmaps" on page 77.

Another kind of GLX drawable is the pixel buffer (or pbuffer), which permits hardware-accelerated off-screen rendering. See "The Pixel Buffer Extension" on page 179.

Rendering Contexts

A rendering context (GLXContext) is an OpenGL data structure that contains the current OpenGL rendering state; an instance of an OpenGL state machine. (For more information, see the section "OpenGL as a State Machine" in Chapter 1, "Introduction to OpenGL," of the *OpenGL Programming Guide*.) Think of a context as a complete description of how to draw what the drawing commands specify.

At most one rendering context can be bound to at most one window or pixmap in a given thread. If a context is bound, it is considered the current context.

OpenGL routines do not specify a drawable or rendering context as parameters. Instead, they implicitly affect the current bound drawable using the current rendering context of the calling thread.

Resources

Resources, in X, are data structures maintained by the server rather than by client programs. Colormaps (as well as windows, pixmaps, and fonts) are implemented as resources.

Rather than keeping information about a window in the client program and sending an entire window data structure from client to server, for instance, window data is stored in the server and given a unique integer ID called an XID. To manipulate or query the window data, the client sends the window's ID number; the server can then perform any requested operation on that window. This reduces network traffic.

Since pixmaps and windows are resources, they are part of the X server and can be shared by different processes (or threads). OpenGL contexts are also resources. In standard OpenGL, they can be shared by threads in the same process but not by separate processes because the API doesn't support this. (Sharing by different processes is possible if the import context extension is supported. See "The Import Context Extension" on page 169.)

Caution: The term "resource" can, in other X-related contexts, refer to items handled by the Resource Manager, items that users can customize for their own use. Don't confuse the two meanings of the word.

Colormaps

A colormap maps pixel values from the framebuffer to intensities on screen. Each pixel value indexes into the colormap to produce intensities of red, green, and blue for display. Depending on hardware limitations, one or more colormaps may be installed at one time, such that windows associated with those maps display with the correct colors. If there is only one colormap, two windows that load colormaps with different values look correct only when they have colormap focus. On all systems, the colormap is a limited resource.

Every X window needs a colormap. If you're using the OpenGL drawing area-widget to render in RGB mode into a TrueColor visual, you may not need to worry about the colormap. In other cases, you may need to install one. For additional information, see "Using Colormaps" on page 69. Colormaps are also discussed in detail in O'Reilly, Volume One.

Libraries, Toolkits, and Tools

This section first discusses programming with widgets and with the Xt (X Toolkit) library, then briefly mentions some other toolkits that facilitate integrating OpenGL with the X Window System.

Widgets and the Xt Library

A widget is a piece of a user interface. Under IRIS IM, buttons, menus, scroll bars, and drawing windows are all widgets.

It usually makes sense to use one of the standard widget sets. A widget set provides a collection of user interface elements that are already combined for easy use; in effect, widgets consisting of several other widgets. It may contain, for example, a simple window with scrollbars, a simple dialog with buttons, and so on. A standard widget set allows you to easily provide a common look and feel for your applications. The two most common widget sets are OSF/Motif and the Athena widget set from MIT.

Silicon Graphics strongly encourages using IRIS IM, the Silicon Graphics port of OSF/Motif, for conformance with Silicon Graphics user interface style and integration with the Indigo Magic desktop. If you use IRIS IM, your application follows the same conventions as other applications on the desktop and is therefore easier to learn and to use.

The examples in this guide use IRIS IM. Using IRIS IM makes it easier to deal with difficult issues such as text management and cut and paste. IRIS IM makes writing complex applications with many user interface components relatively simple. This simplicity doesn't come for free; an application that has minimal user interactions incurs a performance penalty over the same application written in Xlib. For an introduction to Xlib, see "The Xlib Library" on page 5.

The Xt Library

Widgets are built using Xt, the X Toolkit Intrinsics, a library of routines for creating and using widgets. Xt is a "meta" toolkit used to build toolkits like Motif or IRIS IM; you can, in effect, use it to extend the existing widgets in your widget sets. Xt uses a callback-driven programming model. It provides tools for common tasks like input handling and animation and frees you from having to handle a lot of the details of Xlib programming.

Note that in most (but not all) cases, using Xlib is necessary only for colormap manipulation, fonts, and mouse and keyboard handling of events. Otherwise, Xt and IRIS IM are enough, though you may pay a certain performance penalty for using widgets instead of programming directly in Xlib.

For More Information

Standard Xt is discussed in detail in O'Reilly, Volume Four. Standard Motif widgets are discussed in more detail in O'Reilly, Volume Six. See "Background Reading" on page xxv for full bibliographic information and for pointers to additional documents about Motif and IRIS IM.

Other Toolkits and Tools

Silicon Graphics makes several other tools and toolkits available that can greatly facilitate designing your IRIS IM interface. See "RapidApp" on page 3, "Open Inventor" on page 4, and "IRIS ViewKit" on page 5 for more information.

Integrating Your OpenGL Program With IRIS IM

To help you get started, this section presents the simplest possible example program that illustrates how to integrate an OpenGL program with IRIS IM. The program itself is followed by a brief explanation of the steps involved and a more detailed exploration of the steps to follow during integration and setup of your own program.

Window creation and event handling, either using Motif widgets or using the Xlib library directly, are discussed in Chapter 3, "OpenGL and X: Examples."

Simple Motif Example Program

The program in Example 2-1 (*motif/simplest.c)* performs setup, creates a window using a drawing area widget, connects the window with a rendering context, and performs some simple OpenGL rendering (see Figure 2-1).

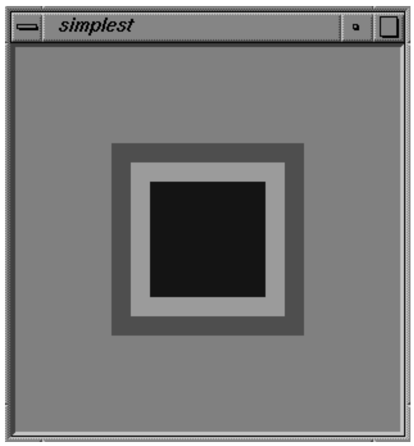


Figure 2-1 Display From simplest.c Example Program

Example 2-1 Simple IRIS IM Program

```
/*
 * simplest - simple single buffered RGBA motif program.
 */
#include <stdlib.h>
#include <stdio.h>
#include <Xm/Frame.h>
#include <X11/GLw/GLwMDrawA.h>
#include <X11/keysym.h>
#include <X11/Xutil.h>
#include <GL/glx.h>

static int      attribs[] = { GLX_RGBA, None};

static String   fallbackResources[] = {
    "*useSchemes: all", “*sgimode:True”,
    "*glxwidget*width: 300", "*glxwidget*height: 300",
    "*frame*shadowType: SHADOW_IN",
    NULL};
```

```
/*Clear the window and draw 3 rectangles*/

void
draw_scene(void) {
   glClearColor(0.5, 0.5, 0.5, 1.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,0.0,0.0);
   glRectf(-.5,-.5,.5,.5);
   glColor3f(0.0,1.0,0.0);
   glRectf(-.4,-.4,.4,.4);
   glColor3f(0.0,0.0,1.0);
   glRectf(-.3,-.3,.3,.3);
   glFlush();
}

/*Process input events*/

static void
input(Widget w, XtPointer client_data, XtPointer call) {
   char buffer[31];
   KeySym keysym;
   XEvent *event = ((GLwDrawingAreaCallbackStruct *) call)->event;

   switch(event->type) {
   case KeyRelease:
      XLookupString(&event->xkey, buffer, 30, &keysym, NULL);
      switch(keysym) {
      case XK_Escape :
         exit(EXIT_SUCCESS);
         break;
      default: break;
      }
      break;
   }
}

/*Process window resize events*/

static void
resize(Widget w, XtPointer client_data, XtPointer call) {
   GLwDrawingAreaCallbackStruct *call_data;
   call_data = (GLwDrawingAreaCallbackStruct *) call;

   glViewport(0, 0, call_data->width, call_data->height);
}
```

```
/*Process window expose events*/

static void
expose(Widget w, XtPointer client_data, XtPointer call) {
    draw_scene();
}

main(int argc, char *argv[]) {
    Display         *dpy;
    XtAppContext     app;
    XVisualInfo     *visinfo;
    GLXContext       glxcontext;
    Widget           toplevel, frame, glxwidget;

    toplevel = XtOpenApplication(&app, "simplest", NULL, 0, &argc,
               argv,fallbackResources, applicationShellWidgetClass,
               NULL, 0);
    dpy = XtDisplay(toplevel);

    frame = XmCreateFrame(toplevel, "frame", NULL, 0);
    XtManageChild(frame);

    /* specify visual directly */
    if (!(visinfo = glXChooseVisual(dpy, DefaultScreen(dpy), attribs)))
        XtAppError(app, "no suitable RGB visual");

    glxwidget = XtVaCreateManagedWidget("glxwidget",
               glwMDrawingAreaWidgetClass, frame, GLwNvisualInfo,
               visinfo, NULL);
    XtAddCallback(glxwidget, GLwNexposeCallback, expose, NULL);
    XtAddCallback(glxwidget, GLwNresizeCallback, resize, NULL);
    XtAddCallback(glxwidget, GLwNinputCallback, input, NULL);

    XtRealizeWidget(toplevel);

    glxcontext = glXCreateContext(dpy, visinfo, 0, GL_TRUE);
    GLwDrawingAreaMakeCurrent(glxwidget, glxcontext);

    XtAppMainLoop(app);
}
```

Looking at the Example Program

As the example program illustrates, integrating OpenGL drawing routines with a simple IRIS IM program involves only a few steps. Except for window creation and event handling, these steps are actually independent of whether the program uses Xt and Motif or Xlib.

The rest of this chapter looks at each step. Each step is discussed in one section:

- “Opening the X Display”
- “Selecting a Visual”
- “Creating a Rendering Context”
- “Creating the Window” (discussed with program examples in “Drawing-Area Widget Setup and Creation” on page 29 and “Creating a Colormap and a Window” on page 42)
- “Binding the Context to the Window”
- “Mapping the Window”

Note that event handling, which is different depending on whether you use Xlib or Motif, is discussed in “Input Handling With Widgets and Xt” on page 34 and, for Xlib programming, “Xlib Event Handling” on page 45.

Opening the X Display

Before making any GLX (or OpenGL) calls, a program must open a display (required) and should find out whether the X server supports GLX (optional).

To open a display, use **XOpenDisplay()** if you’re programming with Xlib, or **XtOpenApplication()** if you’re working with widgets as in Example 2-1 above. **XtOpenApplication()** actually opens the display and performs some additional setup:

- initializing Xt
- opening an X server connection
- creating an X context (not a GLX context) for the application
- creating an application shell widget
- processing command line options
- registering fallback resources

It is recommend (but not required) that you find out whether the X server supports GLX by calling **glXQueryExtension()**.

```
Bool glXQueryExtension ( Display *dpy, int *errorBase, int *eventBase )
```

In most cases, NULL is appropriate for both *errorBase* and *eventBase*. See the reference page for more information.

Note: This call is not required (and therefore not part of *motif/simplest.c*), because **glXChooseVisual()** simply fails if GLX is not supported. It is included here because it's recommended for the sake of portability.

If **glXQueryExtension()** succeeds, use **glXQueryVersion()** to find out which version of GLX is being used; an older version of the extension may not be able to do everything your version can do.

The following pseudo-code demonstrates checking for the version number:

```
glXQueryVersion(dpy, &major, &minor);
if (((major == 1) && (minor == 0)){
     /*assume GLX 1.0, avoid GLX 1.1 functionality*/
     }else
     /*can use GLX 1.1 functionality*/
 }
```

Currently, GLX 1.0 and GLX 1.1 are supported as follows:

GLX 1.0	IRIX 5.1, 5.2, and 6.0.1
GLX 1.1	IRIX 5.3, 6.1, and 6.2

GLX 1.1 supports a few additional functions and provides a mechanism for using extensions. See the *glxintro* reference page.

Selecting a Visual

A visual determines how pixel values are mapped to the screen. The display mode of your OpenGL program (RGBA or color index) determines which X visuals are suitable. To find a visual with the attributes you want, call **glXChooseVisual()** with the desired parameters. Here's the function prototype:

```
XVisualInfo* glXChooseVisual(Display *dpy, int screen, int *attribList)
```

- The first two parameters specify the display and screen. The display was earlier opened with **XtOpenApplication()** or **XOpenDisplay().**

- The third parameter is a list of the attributes you want your visual to have, specified as an array of integers with the special value None as the final element in the array. Attributes specify, for example
  - whether to use RGBA or color-index mode (depending on whether GLX_RGBA is True or False)
  - whether to use double-buffering or not (depending on the value of GLX_DOUBLEBUFFER)
  - how deep the depth buffer should be (depending on the value of GLX_DEPTH_SIZE)

  In Example 2-1 above, the only attribute specified is an RGB display:

  ```
  static int      attribs[] = { GLX_RGBA, None};
  ```

The visual returned by **glXChooseVisual()** is always a visual that supports OpenGL. It is guaranteed to have Boolean attributes matching those specified, and integer attributes with values at least as large as those specified. For detailed information, see the **glXChooseVisual()** reference page.

The framebuffer capabilities and other attributes of a window are determined statically by the visual used to create it. For example, to change a window from single-buffer to double-buffer, you have to switch to a different window created with a different visual.

Note: In general, ask for 1 bit of red, green, and blue to get maximum color resolution. Zero matches to the smallest available color resolution.

Instead of calling **glXChooseVisual()**, you can also choose a visual as follows:

- Ask the X server for a list of all visuals using **XGetVisualInfo()** and then call **glXGetConfig()** to query the attributes of the visuals. Be sure to use a visual for which the attribute GLX_USE_GL is True.
- If you've decided to use IRIS IM, call **XtCreateManagedWidget()**, provide GLwDrawingAreaWidget as the parent, and let the widget choose the visual for you.

There is also an experimental extension that allows you to create and choose a glXFBConfig construct, which packages GLX drawable information, for use instead of a visual. See "The Framebuffer Configuration Extension" on page 171.

Creating a Rendering Context

Creating a rendering context is the application's responsibility. Even if you choose to use IRIS IM, the widget does no context management. Before you can draw anything, you must therefore create a rendering context for OpenGL using **glXCreateContext()**, which has the following function prototype:

```
GLXContext glXCreateContext(Display *dpy, XVisualInfo *vis,
                            GLXContext shareList, Bool direct)
```

Here's how you use the arguments:

dpy	The display you've already opened.
vis	The visual you've chosen with **glXChooseVisual()**.
sharedList	A context to share display lists with, or NULL to not share display lists.
direct	Lets you specify direct or indirect rendering. For best performance, always request direct rendering. The OpenGL implementation automatically switches to indirect rendering when direct rendering isn't possible (for example, when rendering remotely). See "Direct and Indirect Rendering" on page 78.

Creating the Window

After picking a visual and creating a context, you need to create a drawable (window or pixmap) that uses the chosen visual. How you create the drawable depends on whether you use Xlib or Motif calls and is discussed, with program examples, in "Drawing-Area Widget Setup and Creation" on page 29 and "Creating a Colormap and a Window" on page 42.

Binding the Context to the Window

If you're working with Xlib, bind the context to the window by calling **glXMakeCurrent().** Example 3-2 on page 39 is a complete Xlib program and illustrates how the function is used.

If you're working with widgets and have an OpenGL context and a window, bind them together with **GLwDrawingAreaMakeCurrent().** This IRIS IM function is a front end to **glXMakeCurrent()**; it allows you to bind the context to the window without having to know the drawable ID and display.

If **GLwDrawingAreaMakeCurrent()** is successful, subsequent OpenGL calls use the new context to draw on the given drawable. The call fails if the context and the drawable are mismatched; that is, if they were created with different visuals.

Note: Don't make OpenGL calls until the context and window have been bound (made current).

For each thread of execution, at most one context can be bound to at most one window or pixmap.

Note: "The Make Current Read Extension" on page 164 allows you to attach separate read and write drawables to a GLX context.

Mapping the Window

A window can become visible only if it's mapped and all its parent windows are mapped. Note that mapping the window is not directly related to binding it to an OpenGL rendering context, but both need to happen if you want to display an OpenGL application.

Mapping the window or realizing the widget is not synchronous with the call that performs the action. When a window is mapped, the window manager makes it visible if no other actions are specified to happen before. For example, some window managers display just an outline of the window instead of the window itself, letting the user position the window. When the user clicks, the window becomes visible.

If a window is mapped but is not yet visible, you may already set OpenGL state; for example, you may load textures or set colors, but rendering to the window is discarded (this includes rendering to a back buffer if you're doing double-buffering). You need to get an Expose event—if using Xlib—or the expose callback before the window is guaranteed to be visible on the screen. The init callback does not guarantee that the window is visible, only that it exists.

How you map the window on the screen depends on whether you've chosen to create an X window from scratch or use a widget:

- To map a window created with Xlib functions, call **XMapWindow()**.
- To map the window created as a widget, use **XtRealizeWidget()** and **XtCreateManagedChild()**, which perform some additional setup as well. For more information, see the reference pages.

Summary

Table 2-2 summarizes the steps that are needed to integrate an OpenGL program with the X Window System. Note that the GLX functions are usually shared, while other functions differ for IRIS IM or Xlib.

Table 2-2 Integrating OpenGL and X

Step	Using IRIS IM	Using Xlib
"Opening the X Display"	XtOpenApplication	XOpenDisplay
Making sure GLX is supported (optional)	glXQueryExtension glXQueryVersion	glXQueryExtension glXQueryVersion
"Selecting a Visual"	glXChooseVisual	glXChooseVisual
"Creating a Rendering Context"	glXCreateContext	glXCreateContext
"Creating the Window" (see Chapter 3, "OpenGL and X: Examples")	XtVaCreateManagedWidget, with glwMDrawingAreaWidgetClass	XCreateColormap XCreateWindow
"Binding the Context to the Window"	GLwDrawingAreaMakeCurrent	glXMakeCurrent
"Mapping the Window"	XtRealizeWidget	XMapWindow

Additional example programs are provided in Chapter 3, "OpenGL and X: Examples."

Compiling With OpenGL and Related Libraries

This section lists compiler options for individual libraries, then lists groups or libraries typically used together.

Link Lines for Individual Libraries

-lGL OpenGL and GLX routines.

-lX11 Xlib, X client library for X11 protocol generation.

-lXext X Extension library, provides infrastructure for X client side libraries (like OpenGL).

-lGLU	OpenGL utility library.
-lXmu	Miscellaneous utilities library (includes colormap utilities).
-lXt	X toolkit library, infrastructure for widgets.
-lXm	Motif widget set library.
-GLw	OpenGL widgets, Motif and core OpenGL drawing area widgets.
-lXi	X input extension library for using extra input devices.
-limage	RGB file image reading and writing routines.

Link Lines for Groups of Libraries

Minimal OpenGL	-lGL -lXext -lX11
with GLU	-lGLU
with Xmu	-lXmu
with Motif and OpenGL widget	-lGLw -lXm -lXt

Chapter 3

OpenGL and X: Examples

Some aspects of integrating your OpenGL program with the X Window System depend on whether you choose IRIS IM widgets or Xlib. This chapter's main focus is to help you with those aspects by looking at example programs:

- "Using Widgets" on page 27 illustrates how to create a window using IRIS IM drawing-area widgets and how to handle input and other events using callbacks.
- "Using Xlib" on page 39 illustrates how to create a colormap and a window for OpenGL drawing. It also provides a brief discussion of event handling with Xlib.

This chapter also briefly discusses fonts: "Using Fonts and Strings" on page 47 looks at a simple example of using fonts with the **glXUseFont()** function.

Note: All integration aspects that aren't dependent on your choice of Xlib or Motif are discussed in "Integrating Your OpenGL Program With IRIS IM" on page 15 in Chapter 2, "Using OpenGL and X: Getting Started."

Using Widgets

This section explains how to use IRIS IM widgets for creating windows, handling input, and performing other activities the OpenGL part of a program does not deal with. The section discusses the following topics:

- "What are OpenGL Drawing-Area Widgets?"
- "Drawing-Area Widget Setup and Creation"
- "Input Handling With Widgets and Xt"
- "Widget Troubleshooting"

What are OpenGL Drawing-Area Widgets?

Using an OpenGL drawing-area widget facilitates rendering OpenGL into an X window. The widget

- provides an environment for OpenGL rendering, including a visual and a colormap
- provides a set of callback routines for redrawing, resizing, input, and initialization (see “Using Drawing-Area Widget Callbacks” on page 32)

OpenGL provides two drawing-area widgets: GLwMDrawingArea—note the M in the name—for use with IRIS IM (or with OSF/Motif), and GLwDrawingArea for use with any other widget sets. Both drawing-area widgets provide two convenience functions:

- **GLwMDrawingAreaMakeCurrent()** and **GLwDrawingAreaMakeCurrent()**
- **GLwMDrawingAreaSwapBuffers()** and **GLwDrawingAreaSwapBuffers()**

The functions allow you to supply a widget instead of the display and window required by the corresponding GLX functions **glXMakeCurrent()** and **glXSwapBuffers()**.

Because the two widgets are nearly identical, and because IRIS IM is available on all Silicon Graphics systems, this chapter uses only the IRIS IM version, even though most of the information also applies to the general version. Here are some of the distinguishing characteristics of GLwMDrawingArea:

- GLwMDrawingArea understands IRIS IM keyboard traversal (moving around widgets with keyboard keys rather than a mouse), although keyboard traversal is turned off by default.
- GLwMDrawingArea is a subclass of the IRIS IM XmPrimitive widget, not a subclass of the Xt Core widget. It therefore has various defaults such as background and foreground colors. GLwMDrawingArea is **not** derived from the standard Motif drawing-area widget class. (See O’Reilly Volume One or the reference pages for Core and for XmPrimitive for more information.)

  Note that the default background colors provided by the widget are used during X rendering, not during OpenGL rendering, so it’s not advisable to rely on default background rendering from the widget. Even when the background colors are not used directly, **XtGetValues()** can be used to query them to allow the graphics to blend in better with the program.
- GLwMDrawingArea has an IRIS IM style creation function, **GLwCreateMDrawingArea()**; you can also create the widget directly through Xt.

For information specific to GLwDrawingArea, see the reference page.

Drawing-Area Widget Setup and Creation

Most of the steps for writing a program that uses a GLwMDrawingArea widget are already discussed in "Integrating Your OpenGL Program With IRIS IM" on page 15. This section explains how to initialize IRIS IM and how to create the drawing-area widget, using code fragments from the *motif/simplest.c* example program (Example 2-1 on page 16). You learn about

- "Setting Up Fallback Resources"
- "Creating the Widgets"
- "Choosing the Visual for the Drawing-Area Widget"
- "Creating Multiple Widgets With Identical Characteristics"
- "Using Drawing-Area Widget Callbacks"

Setting Up Fallback Resources

This section briefly explains how to work with resources in the context of an OpenGL program. In Xt, resources provide widget properties, allowing you to customize how your widgets will look. Note that the term "resource" used here refers to window properties stored by a resource manager in a resource database, not to the data structures for windows, pixmaps, and context discussed earlier.

Fallback resources inside a program are used when a widget is created and the application can't open the class resource file when it calls **XtOpenApplication()** to open the connection to the X server. (In the code fragment below, the first two resources are specific to Silicon Graphics and give the application a Silicon Graphics look and feel.)

```
static String  fallbackResources[] = {
    "*useSchemes: all",”*sgimode:True”,
    "*glxwidget*width: 300",
    "*glxwidget*height: 300",
    "*frame*shadowType: SHADOW_IN",
    NULL};
```

Note: Applications should ship with resource files installed in a resource directory (in */usr/lib/X11/app-defaults*). If you do install such a file automatically with your application, there is no need to duplicate the resources in your program.

Creating the Widgets

Widgets always exist in a hierarchy, with each widget contributing to what's visible on screen. There's always a top-level widget and almost always a container widget (for example, form or frame). In addition, you may decide to add buttons or scroll bars, which are also part of the IRIS IM widget set. Creating your drawing surface therefore consists of two steps:

1. Create parent widgets, namely the top-level widget and a container widget. *motif/simplest.c,* Example 2-1 on page 16, uses a Form container widget and a Frame widget to draw the 3D box:

   ```
   toplevel = XtOpenApplication(&app, "simplest", NULL, 0, &argc, argv,
             fallbackResources, applicationShellWidgetClass, NULL, 0);
   ...
   form = XmCreateForm(toplevel, "form", args, n);
   XtManageChild(form);
   ....
   frame = XmCreateFrame (form, "frame", args, n);
   ...
   ```

   For more information, see the reference pages for XmForm and XmFrame.

2. Create the GLwMDrawingArea widget itself in either of two ways:

   - Call **GLwCreateMDrawingArea()**. You can specify each attribute as an individual resource or pass in an XVisualInfo pointer obtained with **glXChooseVisual()**. This is discussed in more detail in the next section, "Choosing the Visual for the Drawing-Area Widget."

     ```
     n = 0
     XSetArg(args[n] GLwNvisualinfo, (XtArgVal)visinfo);
     n++;
     glw = GLwCreateMDrawingArea(frame, "glwidget", args, n);
     ```

   - As an alternative, call **XtVaCreateManagedWidget()** and pass it a pointer to the visual you've chosen. In that case, use glwMDrawingAreaWidgetClass as the parent and GLwNvisualInfo to specify the pointer. Here's an example from *motif/simplest.c:*

     ```
     glxwidget = XtVaCreateManagedWidget
                  ("glxwidget", glwMDrawingAreaWidgetClass, frame,
                   GLwNvisualInfo, visinfo, NULL);
     ```

Caution: Creating the widget does not actually create the window. An application must wait until after it has realized the widget before performing any OpenGL operations to the window, or use the ginit callback to indicate when the window has been created.

Note that unlike most other Motif widgets, the OpenGL widget explicitly sets the visual. Once a visual is set and the widget is realized, the visual can no longer be changed.

Choosing the Visual for the Drawing-Area Widget

There are three ways of configuring the GLwMDrawingArea widget when calling the widget creation function, all done through resources:

- Pass in separate resources for each attribute (for example GLwNrgba, GLwNdoublebuffer).
- Pass in an attribute list of the type used by **glXChooseVisual(),** using the GLwNattribList resource.
- Select the visual yourself, using **glXChooseVisual(),** and pass in the returned XvisualInfo as the GLwNvisualInfo resource.

If you wish to provide error handling, call **glXChooseVisual()**, as all the example programs do (although for the sake of brevity, none of the examples actually provides error handling). If you provide the resources and let the widget choose the visual, the widget just prints an error message and quits. Note that a certain visual may be supported on one system but not on another, so appropriate error handling is critical to a robust program.

The advantage of using a list of resources is that you can override them with the *app-defaults* file.

Creating Multiple Widgets With Identical Characteristics

Most applications have one context per widget, though sharing is possible. If you want to use multiple widgets with the same configuration, you must use the same visual for each widget. Windows with different visuals can't share contexts. To share contexts:

1. Extract the GLwNvisualInfo resource from the first widget you create.
2. Use that visual in the creation of subsequent widgets.

Using Drawing-Area Widget Callbacks

The GLwMDrawingArea widget provides callbacks for redrawing, resizing, input, and initialization, as well as the standard XmNdestroyCallback provided by all widgets.

Each callback must first be defined and then added to the widget. In some cases, this is quite simple, as, for example, the resize callback from *motif/simplest.c*:

```
static void
resize(Widget w, XtPointer client_data, XtPointer call) {
   GLwDrawingAreaCallbackStruct *call_data;
   call_data = (GLwDrawingAreaCallbackStruct *) call;

   glViewport(0, 0, call_data->width, call_data->height);
}
```

Other cases are slightly more complex, such as the input callback from *motif/simplest.c*, which exits when the user presses the **<Esc>** key:

```
static void
input(Widget w, XtPointer client_data, XtPointer call) {
    char buffer[31];
    KeySym keysym;
    XEvent *event = ((GLwDrawingAreaCallbackStruct *)call) ->event;

    switch(event->type) {
    case KeyRelease:
       XLookupString(&event->xkey, buffer, 30, &keysym, NULL);
       switch(keysym) {
       case XK_Escape :
              exit(EXIT_SUCCESS);
              break;
       default: break;
       }
       break;
    }
}
```

To add callbacks to a widget, use **XtAddCallback()**; for example:

```
XtAddCallback(glxwidget, GLwNexposeCallback, expose, NULL);
XtAddCallback(glxwidget, GLwNresizeCallback, resize, NULL);
XtAddCallback(glxwidget, GLwNinputCallback, input, NULL);
```

Each callback must ensure that the thread is made current with the correct context to the window associated with the widget generating the callback. You can do this by calling either **GLwMDrawingAreaMakeCurrent()** or **glXMakeCurrent()**.

If you're using only one GLwMDrawingArea, you can call a routine to make the widget "current" just once, after initializing the widget. However, if you're using more than one GLwMDrawingArea or rendering context, you need to make the correct context and the window current for each callback (see "Binding the Context to the Window" on page 22).

The following callbacks are available:

- **GLwNginitCallback**. Specifies the callbacks to be called when the widget is first realized. You can use this callback to perform OpenGL initialization, such as creating a context, since no OpenGL operations can be done before the widget is realized. Callback reason is GLwCR_GINIT.

  Use of this callback is not necessary. Anything done in this callback can also be done after the widget hierarchy has been realized. You can use the callback to keep all the OpenGL code together, keeping the initialization in the same file as the widget creation rather than with widget realization.

  If you create a GLwDrawingArea widget as a child of an already realized widget, it's not possible to add the ginit callback before the widget is realized because the widget is immediately realized at creation. In that case, you should initialize immediately after creating the widget.

- **GLwNexposeCallback**. Specifies the callbacks to be called when the widget receives an Expose event. The callback reason is GLwCR_EXPOSE. The callback structure also includes information about the Expose event. Usually the application should redraw the scene whenever this callback is called.

  Note: An application should not perform any OpenGL drawing until it receives an expose callback, although it may set the OpenGL state; for example, it may create display lists and so on.

- **GLwNinputCallback**. Specifies the callbacks to be called when the widget receives a keyboard or mouse event. The callback structure includes information about the input event. Callback reason is GLwCR_INPUT.

  The input callback is a programming convenience; it provides a convenient way to catch all input events. You can often create a more modular program, however, by providing specific actions and translations in the application rather than using a single catchall callback. See "Input Handling With Widgets and Xt" on page 34 for more information.

- **GLwNresizeCallback**. Specifies the callbacks to be called when the GLwDrawingArea is resized. The callback reason is GLwCR_RESIZE. Normally, programs resize the OpenGL viewport and possibly reload the OpenGL projection matrix (see the *OpenGL Programming Guide*). An expose callback follows. Avoid performing rendering inside the resize callback.

Input Handling With Widgets and Xt

This section explains how to perform input handling with widgets and Xt. It covers:

- "Background Information"
- "Using the Input Callback"
- "Using Actions and Translations"

Background Information

Motif programs are callback driven. They differ in that respect from IRIS GL programs, which implement their own event loops to process events. To handle input with a widget, you can either use the input callback built into the widget or use actions and translations (Xt-provided mechanisms that map keyboard input into user-provided routines). Both approaches have advantages:

- Input callbacks are usually simpler to write, and they're more unified; all input is handled by a single routine that can maintain a private state (see "Using the Input Callback").
- The actions-and-translations method is more modular, because translations have one function for each action. Also, with translations the system does the keyboard parsing so your program doesn't have to do it. Finally, translations allow the user to customize the application's key bindings. See "Using Actions and Translations" on page 36.

Note: To allow smooth porting to other systems, as well as for easier integration of X and OpenGL, always separate event handling from the rest of your program.

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 most applications.

For example, to have the input callback called on all pointer motions, not just on mouse button presses, add the following to the *app-defaults* file:

```
appname*widgetname.translations : \
    <KeyDown>:      glwInput() \n\
    <KeyUp>:        glwInput() \n\
    <BtnDown>:      glwInput() \n\
    <BtnUp>:        glwInput() \n\
    <BtnMotion>:    glwInput() \n\
    <PtrMoved>:     glwInput()
```

The callback is passed an X event. It interprets the X events and performs the appropriate action. It's your application's responsibility to interpret the event—for example, to convert an X keycode into a key symbol—and to decide what to do with it.

Example 3-1 is from *motif/mouse.c*, a double-buffered RGBA program that uses mouse motion events.

Example 3-1 Motif Program That Handles Mouse Events

```
static void
input(Widget w, XtPointer client_data, XtPointer call) {
   char buffer[31];
   KeySym keysym;
   XEvent *event = ((GLwDrawingAreaCallbackStruct *) call)->event;
   static mstate, omx, omy, mx, my;

   switch(event->type) {
   case KeyRelease:
      XLookupString(&event->xkey, buffer, 30, &keysym, NULL);
      switch(keysym) {
      case XK_Escape:
         exit(EXIT_SUCCESS);
         break;
      default: break;
      }
      break;
    case ButtonPress:
        if (event->xbutton.button == Button2) {
            mstate |= 2;
            mx = event->xbutton.x;
            my = event->xbutton.y;
```

```
        } else if (event->xbutton.button == Button1) {
            mstate |= 1;
            mx = event->xbutton.x;
            my = event->xbutton.y;
        }
        break;
    case ButtonRelease:
        if (event->xbutton.button == Button2)
            mstate &= ~2;
        else if (event->xbutton.button == Button1)
            mstate &= ~1;
        break;
    case MotionNotify:
        if (mstate) {
            omx = mx;
            omy = my;
            mx = event->xbutton.x;
            my = event->xbutton.y;
            update_view(mstate, omx,mx,omy,my);
        }
        break;
   }
```

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 you press the <Esc> key, the exit routine **quit()** is called
- when you press the left mouse button, rotation occurs
- when you press f, the program zooms in

The translations need to be combined with an action task that maps string names like quit() to real function pointers. Below is an example of a translation table:

```
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()
```

When you press the left mouse button, the **start_rotate()** action is called; when it is released, the **stop_rotate()** action is called.

The last entry is a little cryptic. It actually says that when the user presses the <Esc> key, **quit()** is called. However, OSF 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 or OSF/Motif, replace <KeyUp>osfCancel with <KeyUp><Esc>.

The translation is only half of what it takes to set up this binding. Although the translation table above contains what look like function names, they're really action names. Your program must also create an action table to bind the action names to actual functions in the program.

For more information on actions and translations, see O'Reilly, *X Toolkit Intrinsics Programming Manual* (Volume 4), most notably Chapter 4, "An Example Application," and Chapter 8, "Events, Translations, and Accelerators." You can view this manual online using IRIS InSight.

Creating Colormaps

By default, a widget creates a colormap automatically. For many programs, this is sufficient. However, it is occasionally necessary to create a colormap explicitly, especially when using color index mode. See "Creating a Colormap and a Window" on page 42 and "Using Colormaps" on page 69 for more information.

Widget Troubleshooting

This section provides troubleshooting information by discussing some common pitfalls when working with widgets.

Note: Additional debugging information is provided in "Tips for Debugging Graphics Programs" on page 205.

Keyboard Input Disappears

A common problem in IRIS IM programs is that keyboard input disappears. This is caused by how IRIS IM handles keyboard focus. When a widget hierarchy has keyboard focus, only one component of the hierarchy receives the keyboard events. The keyboard input might be going to the wrong widget.

There are two solutions to this:

- The easiest solution is to set the resource

  ```
  keyboardFocusPolicy: POINTER
  ```

  for the application. This overrides the default traversal method (explicit traversal) where you can select widgets with keyboard keys rather than the mouse so that input focus follows the pointer only. The disadvantages of this method are that it eliminates explicit traversal for users who prefer it and it forces a nondefault model.

- A better solution is to set the resource

  ```
  *widget.traversalOn: TRUE
  ```

  where *widget* is the name of the widget, and to call

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

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

Inheritance Issues

In Xt, shell widgets, which include top-level windows, popup windows, and menus,

- inherit their colormap and depth from their parent widget
- inherit their visual from the parent window

If the visual does not match the colormap and depth, this leads to a BadMatch X protocol error.

In a typical IRIS IM program, everything runs in the default visual, and the inheritance from two different places does not cause problems. However, when a program uses both OpenGL and IRIS IM, it requires multiple visuals, and you have to be careful. Whenever you create a shell widget as a child of a widget in a non-default visual, specify pixel depth, colormap, and visual for that widget explicitly. This happens with menus or popup windows that are children of OpenGL widgets. See “Using Popup Menus With the GLwMDrawingArea Widget” on page 64.

If you do get a BadMatch error, follow these steps to determine its cause:

1. Run the application under a C debugger, such as dbx or cvd (the Case Vision debugger) with the **-sync** flag.

   The **-sync** flag tells Xt to call **XSynchronize()**, forcing all calls to be made synchronously. If your program is not Xt-based, or if you aren’t using standard argument parsing, call **XSynchronize(display, TRUE)** directly inside your program.

2. Using the debugger, set a breakpoint in **exit()** and run the program.

   When the program fails, you have a stack trace you can use to determine the problem.

Note: If you don’t use the **-sync** option, the stack dump on failure is meaningless: X batches multiple requests and the error is delayed.

Using Xlib

This section explains how to use Xlib for creating windows, handling input, and performing other activities the OpenGL part of a program does not deal with. Since the complete example program in Chapter 2, “Using OpenGL and X: Getting Started” used widgets, this section starts with a complete annotated example program for Xlib, so you have both available as needed. After that, you learn about

- Creating a Colormap and a Window
- Xlib Event Handling

Simple Xlib Example Program

Example 3-2 lists the complete *Xlib/simplest.c* example program.

Example 3-2 Simple Xlib Example Program

```
/*
 * simplest - simple single buffered RGBA xlib program.
 */
/* compile: cc -o simplest simplest.c -lGL -lX11 */
```

```
#include <GL/glx.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>

static int attributeList[] = { GLX_RGBA, None };

static void
draw_scene(void) {
    glClearColor(0.5, 0.5, 0.5, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0,0.0,0.0);
    glRectf(-.5,-.5,.5,.5);
    glColor3f(0.0,1.0,0.0);
    glRectf(-.4,-.4,.4,.4);
    glColor3f(0.0,0.0,1.0);
    glRectf(-.3,-.3,.3,.3);
    glFlush();
}

static void
process_input(Display *dpy) {
    XEvent event;
    Bool redraw = 0;

    do {
        char buf[31];
        KeySym keysym;

        XNextEvent(dpy, &event);
        switch(event.type) {
        case Expose:
            redraw = 1;
            break;
        case ConfigureNotify:
            glViewport(0, 0, event.xconfigure.width,
                    event.xconfigure.height);
            redraw = 1;
            break;
        case KeyPress:
            (void) XLookupString(&event.xkey, buf, sizeof(buf),
                    &keysym, NULL);
            switch (keysym) {
```

```
                case XK_Escape:
                    exit(EXIT_SUCCESS);
                default:
                    break;
                }
            default:
                break;
            }
        } while (XPending(dpy));
        if (redraw) draw_scene();
}

static void
error(const char *prog, const char *msg) {
    fprintf(stderr, "%s: %s\n", prog, msg);
    exit(EXIT_FAILURE);
}
int
main(int argc, char **argv) {
    Display *dpy;
    XVisualInfo *vi;
    XSetWindowAttributes swa;
    Window win;
    GLXContext cx;

    /* get a connection */
    dpy = XOpenDisplay(0);
    if (!dpy) error(argv[0], "can't open display");

    /* get an appropriate visual */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributeList);
    if (!vi) error(argv[0], "no suitable visual");

    /* create a GLX context */
    cx = glXCreateContext(dpy, vi, 0, GL_TRUE);

    /* create a colormap */
    swa.colormap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                                   vi->visual, AllocNone);
    /* create a window */
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | StructureNotifyMask | KeyPressMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 300,
                        300, 0, vi->depth, InputOutput, vi->visual,
                        CWBorderPixel|CWColormap|CWEventMask, &swa);
```

```
    XStoreName(dpy, win, "simplest");
    XMapWindow(dpy, win);

    /* connect the context to the window */
    glXMakeCurrent(dpy, win, cx);

    while (1) process_input(dpy);
}
```

Creating a Colormap and a Window

A colormap determines the mapping of pixel values in the framebuffer to color values on the screen. Colormaps are created with respect to a specific visual.

When you create a window, you must supply a colormap for it. The visual associated with a colormap must match the visual of the window using the colormap. Most X programs use the default colormap because most X programs use the default visual. The easiest way to obtain the colormap for a particular visual is to call **XCreateColormap()**:

```
Colormap XCreateColormap (Display *display, Window w, Visual *visual,
                          int alloc)
```

Here's how Example 3-2 calls **XCreateColormap()**:

```
swa.colormap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                                    vi->visual, AllocNone);
```

The parameters specify the display, window, and visual, and the number of colormap entries to allocate. The *alloc* parameter can have the special value AllocAll or AllocNone. While it is easy to simply call **XCreateColormap()**, you are encouraged to share colormaps. See Example 4-2 on page 71 for details on how to do this.

Note that you can't use AllocAll if the colormap corresponds to a visual that has transparent pixels, because the colormap cell that corresponds to the transparent pixel cannot be allocated with AllocAll. For more information about colormaps, see "Using Colormaps" on page 69. For information on overlays, which use a visual with a transparent pixel, see "Using Overlays" on page 58.

You can then create a window using **XCreateWindow()**. Before calling **XCreateWindow()**, set the attributes you want in the *attributes* variable. When you make the call, indicate *valuemask* by OR-ing together symbolic constants that specify the attributes you've set. Here's how Example 3-2 does it:

```
swa.background_pixmap = None;
swa.border_pixel = 0;
swa.event_mask = ExposureMask | StructureNotifyMask | KeyPressMask;
win = XCreateWindow(
             dpy,                                /*display*/
             RootWindow(dpy, vi->screen),        /*parent*/
             0,                                  /*x coordinate*/
             0,                                  /*y coordinate*/
             300,                                /*width*/
             300,                                /*height*/
             0,                                  /*border width*/
             vi->depth,                          /*depth*/
             InputOutput,                        /*class*/
             vi->visual,                         /*visual*/
             CWBackPixmap|CWBorderPixel|CWColormap|CWEventMask,
                                                 /*valuemask*/
             &swa                                /*attributes*/
);
```

Most of the parameters are self-explanatory. Here are three that are not:

- *class* indicates whether the window is InputOnly or InputOutput.

  Note: InputOnly windows cannot be used with GLX contexts.

- *valuemask* specifies which window attributes are provided by the call.
- *attributes* specifies the settings for the window attributes. The XSetWindowAttributes structure contains a field for each of the allowable attributes.

Note: If the window's visual or colormap does not match the visual or colormap of the window's parent, you **must** specify a border pixel to avoid a BadMatch X protocol error. Most windows specify a border zero pixels wide, so the value of the border pixel is unimportant; zero works fine.

If the window you are creating is a top-level window (meaning it was created as a child of the root window), consider calling **XSetWMProperties()** to set the window's properties after you've created it.

```
void XSetWMProperties(Display *display, Window w,
                     XTextProperty *window_name, XTextProperty *icon_name,
                     char **argv, int argc, XSizeHints *normal_hints,
                     XWMHints *wm_hints, XClassHint *class_hints)
```

XSetWMProperties() provides a convenient interface for setting a variety of important window properties at once. It merely calls a series of other property-setting functions, passing along the values you pass in. For more information, see the reference page.

Note that two useful properties are the window name and the icon name. The example program calls **XStoreName()** instead to set the window and icon names.

Installing the Colormap

Applications should rely on the window manager to install the colormaps instead of calling **XInstallColormap()** directly. The window manager automatically installs the appropriate colormaps for a window, whenever that window gets keyboard focus.

By default, the window manager looks at the top-level window of a window hierarchy and installs that colormap when the window gets keyboard focus. For a typical X-based application, this is sufficient, but an application based on OpenGL typically uses multiple colormaps: the top-level window uses the default X colormap, and the Open GL window uses a colormap suitable for OpenGL.

To address this multiple colormap issue, call the function **XSetWMColormapWindows()** passing the display, the top-level window, a list of windows whose colormaps should be installed, and the number of windows in the list.

The list of windows should include one window for each colormap, including the top-level window's colormap (normally represented by the top-level window). For a typical OpenGL program that does not use overlays, the list contains two windows: the OpenGL window and the top-level window. The top-level window should normally be last in the list. Xt programs may use **XtSetWMColormapWindows()** instead of **XSetWMColormapWindows()**, which uses widgets instead of windows.

Note: The program must call **XSetWMColormapWindows()** even if it is using a TrueColor visual. Some hardware simulates TrueColor through the use of a colormap. Even though the application doesn't interact with the colormap directly, it is still there. If you don't call **XSetWMColormapWindows(),** your program may run correctly only some of the time, and only on some systems.

Use the *xprop* program to determine whether **XSetWMColormapWindows()** was called. Click the window and look for the WM_COLORMAP_WINDOWS property. This should be a list of the windows. The last one should be the top-level window. Use *xwininfo,* providing the ID of the window as an argument, to determine what colormap the specified window is using, and whether that colormap is installed.

Xlib Event Handling

This section discusses different kinds of user input and explains how you can use Xlib to perform them. OpenGL programs running under the X Window System are responsible for responding to events sent by the X server. Examples of X events are Expose, ButtonPress, ConfigureNotify, and so on.

Note: In addition to mouse devices, Silicon Graphics systems support various other input devices (for example, spaceballs). You can integrate them with your OpenGL program using the X input extension. For more information, see the *X Input Extension Library Specification* available online through IRIS Insight.

Handling Mouse Events

To handle mouse events, your program first has to request them, then use them in the main (event handling) loop. Here's an example code fragment from *Xlib/mouse.c*, an Xlib program that uses mouse motion events. Example 3-3 shows how the mouse processing, along with the other event processing, is defined.

Example 3-3 Event Handling With Xlib

```
static int
process_input(Display *dpy) {
    XEvent event;
    Bool redraw = 0;
    static int mstate, omx, omy, mx, my;

    do {
        char buf[31];
        KeySym keysym;

        XNextEvent(dpy, &event);
        switch(event.type) {
        case Expose:
            redraw = 1;
            break;
        case ConfigureNotify:
            glViewport(0, 0, event.xconfigure.width,
                       event.xconfigure.height);
            redraw = 1;
            break;
```

```
        case KeyPress:
            (void) XLookupString(&event.xkey, buf, sizeof(buf),
                   &keysym, NULL);
            switch (keysym) {
            case XK_Escape:
                exit(EXIT_SUCCESS);
            default:
                break;
            }
        case ButtonPress:
            if (event.xbutton.button == Button2) {
                mstate |= 2;
                mx = event.xbutton.x;
                my = event.xbutton.y;
            } else if (event.xbutton.button == Button1) {
                mstate |= 1;
                mx = event.xbutton.x;
                my = event.xbutton.y;
            }
            break;
        case ButtonRelease:
            if (event.xbutton.button == Button2)
                mstate &= ~2;
            else if (event.xbutton.button == Button1)
                mstate &= ~1;
            break;
        case MotionNotify:
            if (mstate) {
                omx = mx;
                omy = my;
                mx = event.xbutton.x;
                my = event.xbutton.y;
                update_view(mstate, omx,mx,omy,my);
                redraw = 1;
            }
            break;
        default:
            break;
        }
    } while (XPending(dpy));
    return redraw;
}
```

The **process_input()** function is then used by the main loop:

```
 while (1) {
        if (process_input(dpy)) {
            draw_scene();
            ...
           }
}
```

Exposing a Window

When a user selects a window that has been completely or partly covered, the X server generates one or more Expose events. It is difficult to determine exactly what was drawn in the now-exposed region and redraw only that portion of the window. Instead, OpenGL programs usually just redraw the entire window. (Note that backing store is not supported on Silicon Graphics systems.)

If redrawing is not an acceptable solution, the OpenGL program can do all your rendering into a GLXPixmap instead of directly to the window; then, any time the program needs to redraw the window, you can simply copy the GLXPixmap's contents into the window using **XCopyArea()**. For more information, see "Using Pixmaps" on page 77.

Caution: Rendering to a GLXPixmap is much slower than rendering to a window. For example, on a RealityEngine, rendering to a pixmap is perhaps 5% the speed of rendering to a window.

When handling X events for OpenGL programs, remember that Expose events come in batches. When you expose a window that is partly covered by two or more other windows, two or more Expose events are generated, one for each exposed region. Each one indicates a simple rectangle in the window to be redrawn. If you are going to redraw the entire window, read the entire batch of Expose events. It is wasteful and inefficient to redraw the window for each Expose event.

Using Fonts and Strings

The simplest approach to text and font handling in GLX is using the **glXUseXFont()** function together with display lists. This section shows you how to use the function by providing an example program. Note that this information is relevant regardless of whether you use widgets or program in Xlib.

The advantage of **glXUseXFont()** is that bitmaps for X glyphs in the font match exactly what OpenGL draws. This solves the problem of font matching between X and OpenGL display areas in your application.

To use display lists to display X bitmap fonts, your code should do the following:

1. Use X calls to load information about the font you want to use.
2. Generate a series of display lists using **glXUseXFont()**, one for each glyph in the font.

   The **glXUseXFont()** function automatically generates display lists (one per glyph) for a contiguous range of glyphs in a font.
3. To display a string, use **glListBase()** to set the display list base to the base for your character series. Then pass the string as an argument to **glCallLists()**.

   Each glyph display list contains a **glBitmap()** call to render the glyph and update the current raster position based on the glyph's width.

The example code fragment provided in Example 3-4 prints the string "The quick brown fox jumps over a lazy dog" in Times Medium. It also prints the entire character set, from ASCII 32 to 127.

Note: You can also use the glc library, which sits atop of OpenGL, for fonts and strings. The library is not specific to GLX and lets you do more than **glXUseXFont().**

Example 3-4 Font and Text Handling

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

GLuint base;

void makeRasterFont(Display *dpy)
{
    XFontStruct *fontInfo;
    Font id;
    unsigned int first, last;
    fontInfo = XLoadQueryFont(dpy,
        "-adobe-times-medium-r-normal--17-120-100-100-p-88-iso8859-1");
```

```
if (fontInfo == NULL) {
        printf ("no font found\n");
        exit (0);
    }

    id = fontInfo->fid;
    first = fontInfo->min_char_or_byte2;
    last = fontInfo->max_char_or_byte2;

    base = glGenLists(last+1);
    if (base == 0) {
        printf ("out of display lists\n");
    exit (0);
    }
    glXUseXFont(id, first, last-first+1, base+first);
}

void printString(char *s)
{
    glListBase(base);
    glCallLists(strlen(s), GL_UNSIGNED_BYTE, (unsigned char *)s);
}

void display(void)
{
    GLfloat white[3] = { 1.0, 1.0, 1.0 };
    long i, j;
    char teststring[33];

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3fv(white);
    for (i = 32; i < 127; i += 32) {
        glRasterPos2i(20, 200 - 18*i/32);
        for (j = 0; j < 32; j++)
            teststring[j] = i+j;
        teststring[32] = 0;
        printString(teststring);
    }
    glRasterPos2i(20, 100);
    printString("The quick brown fox jumps");
    glRasterPos2i(20, 82);
    printString("over a lazy dog.");
    glFlush ();
}
```

Chapter 4

OpenGL and X: Advanced Topics

This chapter helps you integrate your OpenGL program with the X Window System by discussing several advanced topics. While understanding the techniques and concepts discussed here is not relevant for all applications, it is important that you master them for certain special situations. The chapter covers the following topics:

Using Animations

Animation in its simplest form consists of drawing an image, clearing it, and drawing a new, slightly different one in its place. However, attempting to draw into a window while that window is being displayed can cause problems such as flickering. The solution is double buffering.

This section discusses double-buffered animation inside an X Window System environment, providing example code as appropriate. You learn about

- "Swapping Buffers"
- "Controlling an Animation With Workprocs"
- "Controlling an Animation With Timeouts"

Xt provides two mechanisms that are suited for continuous animation:

- "Controlling an Animation With Workprocs" on page 53 results in the fastest animation possible. If you use workprocs, the program swaps buffers as fast as possible; which is useful if rendering speed is variable enough that constant speed animation is not possible. Workproc animations also give other parts of the application priority. The controls don't become less responsive just because the animation is being done. The cost of this is that the animation slows down or may stop when the user brings up a menu or uses other controls.
- "Controlling an Animation With Timeouts" on page 56 results in a constant speed animation. Animations that use timeouts compete on even footing with other Xt events; the animation won't stop because the user interacts with other components of the animation.

Note: Controlling animations with workprocs and timeouts applies only to Xt-based programs.

Swapping Buffers

A double-buffered animation displays one buffer while drawing into another (undisplayed) buffer, then swaps the displayed buffer with the other. In OpenGL, the displayed buffer is called the front buffer, and the undisplayed buffer is called the back buffer. This sort of action is common in OpenGL programs; however, swapping buffers is a window-related function, not a rendering function, so you can't do it directly with OpenGL.

To swap buffers, use **glXSwapBuffers()** or (when using the widget) the convenience function **GLwDrawingAreaSwapBuffers()**. The **glXSwapBuffers()** function takes a display and a window as input—pixmaps don't support buffer swapping—and swaps the front and back buffers in the drawable. All renderers bound to the window in question continue to have the correct idea of which is the front buffer and which the back buffer. Note that once you call **glXSwapBuffers()**, any further drawing to the given window is suspended until after the buffers have been swapped.

Silicon Graphics systems support hardware double buffering; this means buffer swap is instantaneous during the vertical retrace of the monitor. As a result, there are no tearing artifacts; that is, you don't simultaneously see part of one buffer and part of the next.

Caution: If the window's visual allows only one color buffer, or if the GLX drawable is a pixmap, **glXSwapBuffers()** has no effect (and generates no error).

There's no need to worry about which buffer the X server draws into if you're using X drawing functions as well as OpenGL; the X server draws only to the current front buffer, and prevents any program from swapping buffers while such drawing is going on. Using the X double buffering extension (DBE), it is possible to render X into the back buffer. DBE is not supported in releases preceding IRIX 6.2.

Note that users like uniform frame rates such as 60 Hz, 30 Hz, or 20 Hz. Animation may otherwise look jerky. A slower consistent rate is therefore preferable to a faster but inconsistent rate. For additional information about optimizing frame rates, see "Optimizing Frame Rate Performance" on page 215. See "The Swap Control Extension" on page 168 to learn how to set a minimum period of buffer swaps.

Controlling an Animation With Workprocs

A workproc (work procedure) is a procedure that Xt calls when the application is idle. The application registers workprocs with Xt and unregisters them when it's time to stop calling them.

Note that workprocs do not provide constant speed animation but animate as fast as the application can.

General Workproc Information

Workprocs can be used to carry out a variety of useful tasks: animation, setting up widgets in the background (to improve application startup time), keeping a file up to date, and so on.

It's important that a workproc not take very long to execute. While a workproc is running, nothing else can run, and the application may appear sluggish or may even appear to hang.

Workprocs return Booleans. To set up a function as a workproc, first prototype the function, then pass its name to **XtAppAddWorkProc()**. Xt then calls the function whenever there's idle time while Xt is waiting for an event. If the function returns True, it's removed from the list of workprocs; if it returns False, it's kept on the list and called again when there's idle time.

To explicitly remove a workproc, call **XtRemoveWorkProc()**. Here are the prototypes for the add and remove functions:

```
XtWorkProcId XtAppAddWorkProc(XtAppContext app_context,
                              XtWorkProc proc, XtPointer client_data)

void XtRemoveWorkProc(XtWorkProcId id)
```

The *client_data* parameter for **XtAppAddWorkProc()** lets you pass data from the application into the workproc, similar to the equivalent parameter used in setting up a callback.

Workproc Example

This section illustrates using workprocs. The example, *motif/animate.c,* is a simple animation driven by a workproc. When the user selects "animate" from the menu, the workproc is registered, as follows:

```
static void
menu(Widget w, XtPointer clientData, XtPointer callData) {
    int entry = (int) clientData;

    switch (entry) {
    case 0:
        if (state.animate_wpid) {
            XtRemoveWorkProc(state.animate_wpid);
            state.animate_wpid = 0;
        } else {
            /* register workproc */
            state.animate_wpid = XtAppAddWorkProc(state.appctx,
                                          redraw_proc, &state.glxwidget);
        }
        break;
    case 1:
        exit(EXIT_SUCCESS);
        break;
    default:
        break;
    }
}
```

The workproc starts executing if the window is mapped (that is, it could be visible but it may be overlapped):

```
static void
map_change(Widget w, XtPointer clientData, XEvent *event, Boolean
                                                             *cont) {
    switch (event->type) {
```

```
    case MapNotify:
    /* resume animation if we become mapped in the animated state */
        if (state.animate_wpid != 0)
             state.animate_wpid = XtAppAddWorkProc(state.appctx,
                                           redraw_proc, &state.glxwidget);
        break;
    case UnmapNotify:
    /* don't animate if we aren't mapped */
        if (state.animate_wpid) XtRemoveWorkProc(state.animate_wpid);
        break;
    }
}
```

If the window is mapped, the workproc calls **redraw_proc()**:

```
static Boolean
redraw_proc(XtPointer clientData) {
    Widget *w = (Widget *)clientData;
    draw_scene(*w);
    return False;
    /*call the workproc again as possible*/
}
```

The **redraw_proc()** function, in turn, calls **draw_scene()**, which swaps the buffers. Note that this program doesn't use **glXSwapBuffers()**, but instead the convenience function **GLwDrawingAreaSwapBuffers()**.

```
static void
draw_scene(Widget w) {
    static float rot = 0.;

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(.1, .1, .8);
    glPushMatrix();
    if ((rot += 5.) > 360.) rot -= 360.;
    glRotatef(rot,0.,1.,0.);
    cube();
    glScalef(0.3,0.3,0.3);
    glColor3f(.8, .8, .1);
    cube();
    glPopMatrix();
    GLwDrawingAreaSwapBuffers(w);
}
```

Note: If an animation is running and the user selects a menu command, the event handling for the command and the animation may end up in a race condition.

Controlling an Animation With Timeouts

The program that performs an animation using timeouts is actually quite similar to the one using workprocs. The main difference is that the timeout interval has to be defined and functions that relied on the workproc now have to be defined to rely on the timeout. Note especially that **redraw_proc()** has to register a new timeout each time it's called.

You may find it most helpful to compare the full programs using *xdiff* or a similar tool. This section briefly points out the main differences between two example programs.

- The redraw procedure is defined to have an additional argument, an interval ID.

  work_animate: `static Boolean redraw_proc(XtPointer clientData);`

  time_animate:
  ```
  static Boolean redraw_proc(XtPointer clientData,
                             XtIntervalId *id);
  ```

- In *time_animate,* a timeout has to be defined; the example chooses 10 ms:

  ```
  #define TIMEOUT 10 /*timeout in milliseconds*/
  ```

- In the state structure, which defines the global UI variables, the interval ID instead of the workproc ID is included.

  work_animate:

  ```
  static struct {        /* global UI variables; keep them together */
      XtAppContext appctx;
      Widget glxwidget;
      Boolean direct;
      XtWorkProcId animate_wpid;
  } state;
  ```

  time_animate:

  ```
  static struct {        /* global UI variables; keep them together */
      XtAppContext appctx;
      Widget glxwidget;
      Boolean direct;
      XtIntervalId animate_toid;
  } state;
  ```

- The **menu()** function and the **map_change()** function are defined to remove or register the timeout instead of the workproc. Here are the two **menu()** functions as an example:

  work_animate:

```
static void
menu(Widget w, XtPointer clientData, XtPointer callData) {
    int entry = (int) clientData;

    switch (entry) {
    case 0:
        if (state.animate_wpid) {
            XtRemoveWorkProc(state.animate_wpid);
            state.animate_wpid = 0;
        } else {
            /* register work proc */
            state.animate_wpid = XtAppAddWorkProc(state.appctx,
                                     redraw_proc, &state.glxwidget);
        }
        break;
    case 1:
        exit(EXIT_SUCCESS);
        break;
    default:
        break;
    }
}
```

  time_animate

```
static void
menu(Widget w, XtPointer clientData, XtPointer callData) {
    int entry = (int) clientData;

    switch (entry) {
    case 0:
        if (state.animate_toid) {
            XtRemoveTimeOut(state.animate_toid);
            state.animate_toid = 0;
        } else {
            /* register timeout */
            state.animate_toid = XtAppAddTimeOut(state.appctx,
                             TIMEOUT, redraw_proc, &state.glxwidget);
        }
        break;
```

```
        case 1:
            exit(EXIT_SUCCESS);
            break;
        default:
            break;
        }
    }
```

- The **redraw_proc()** function has to register a new timeout each time it's called. Note that this differs from the workproc approach, where the application automatically continues animating as long as the system isn't doing something else.

```
static void
redraw_proc(XtPointer clientData, XtIntervalId *id) {
    Widget *w = (Widget *)clientData;
    draw_scene(*w);
    /* register a new timeout */
    state.animate_toid = XtAppAddTimeOut(state.appctx, TIMEOUT,
                                     redraw_proc, &state.glxwidget);
}
```

Using Overlays

Overlays are useful in situations where you want to preserve an underlying image while displaying some temporary information. Examples for this are popup menus, annotations, or rubber-banding. This section explains overlays and shows you how to them, discussing the following topics:

- "Introduction to Overlays"
- "How to Create Overlays"
- "Rubber Banding"

Introduction to Overlays

An overlay plane is a set of bitplanes displayed preferentially to the normal planes. Non-transparent pixels in the overlay plane are displayed in preference to the underlying pixels in the normal planes. Windows in the overlay planes do not damage windows in the normal plane.

If you have something in the main window that's fairly expensive to draw into and want to have something else on top, such as an annotation, you can use a transparent overlay plane to avoid redrawing the more expensive main window. Overlays are well-suited for popup menus, dialog boxes, and "rubber-band" image resizing rectangles. You can also use overlay planes for text annotations floating "over" an image and for certain transparency effects.

Note: Transparency discussed here is distinct from alpha buffer blending transparency effects. See the section "Blending" in Chapter 7, "Blending, Anti-Aliasing, and Fog," in the *OpenGL Programming Guide.*

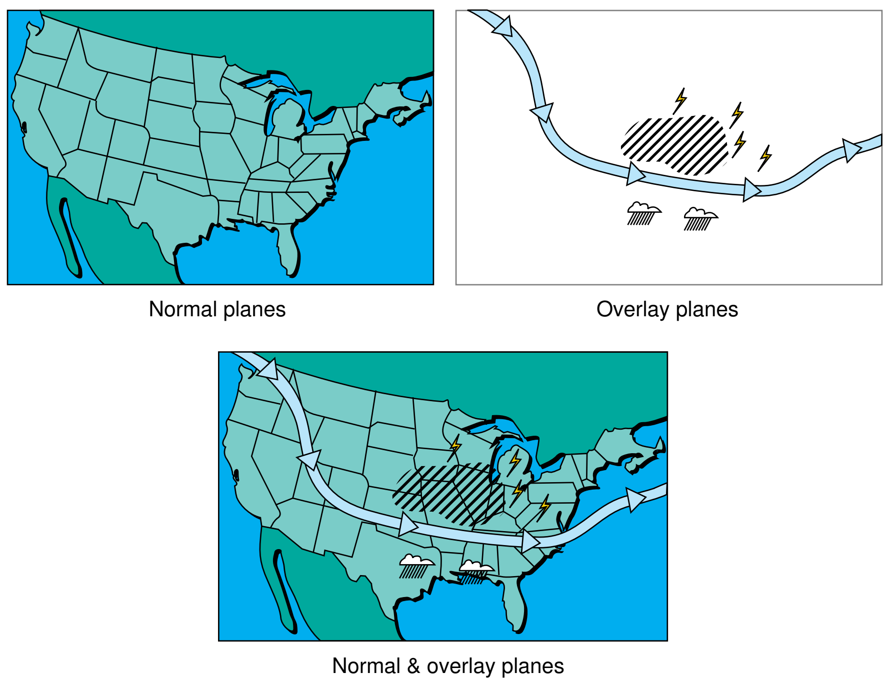


Figure 4-1 Overlay Plane Used for Transient Information

A special value in the overlay planes indicates transparency. On Silicon Graphics systems, it's always the value zero. Any pixel with the value zero in the overlay plane is not painted, allowing the color of the corresponding pixel in the normal planes to show.

The concepts discussed in this section apply more generally to any number of framebuffer layers, for example, underlay planes (which are covered up by anything in equivalent regions of higher-level planes).

You can use overlays in two ways:

- To draw additional graphics in the overlay plane on top of your normal plane OpenGL widget, create a separate GLwMDrawingArea widget in the overlay plane and set the GLX_LEVEL resource to 1. Position the overlay widget on top of the normal plane widget.

  Note that since the GLwMDrawingArea widget is not a manager widget, it is necessary to create both the normal and overlay widgets as children of some manager widget—for example, a form—and have that widget position the two on top of each other. Once the windows are realized, you must call **XRaiseWindow()** to guarantee that the overlay widget is on top of the normal widget. Code fragments in "How to Create Overlays" on page 60 illustrate this. The whole program is included as *overlay.c* in the source tree.
- To create menus, look at examples in */usr/src/X11/motif/overlay_demos*. They are present if you have the *motif_dev.sw.demo* subsystem installed. Placing the menus in the overlay plane avoids the need for expensive redrawing of the OpenGL window underneath them. While the demos do not deal specifically with OpenGL, they do show how to place menus in the overlay plane.

A Note for IRIS GL Users

IRIS GL supports the concept of popup planes, which are one level higher than the default overlay plane. Drawing in the popup planes in IRIS GL doesn't necessarily require a window, but you can't count on avoiding damage to anything non-transient drawn in those planes (for example, objects drawn by other applications).

When working with OpenGL and the X Window System, the situation is different: You have to create a separate window for any overlay rendering. Currently, no OpenGL implementation on a Silicon Graphics system supports a level greater than one.

How to Create Overlays

This section explains how to create overlay planes, using an example program based on Motif. If you create the window using Xlib, the same process is valid (and a parallel example program is available in the example program directory).

The example program from which the code fragments are taken, *motif/overlay.c,* uses the visual info extension to find a visual with a transparent pixel. See “The Visual Info Extension” on page 165 for more information.

Note: This example does not work if the visual info extension is not available (see “How to Check for OpenGL Extension Availability” on page 82). The visual info extension is available only in IRIX 6.2. In IRIX 5.3 and earlier releases, you must look at the TRANSPARENT_OVERLAYS property on the root window to get the information.

To create the overlay, follow these steps:

1. Define attribute lists for the two widgets (the window and the overlay). For the overlay, specify GLX_LEVEL as 1 and GLX_TRANSPARENT_TYPE_EXT as GLX_TRANSPARENT_RGB_EXT if the visual info extension is available.

```
static int attribs[] = { GLX_RGBA, GLX_DOUBLEBUFFER, None};
static int ov_attribs[] = {
                  GLX_BUFFER_SIZE, 2,
                  GLX_LEVEL, 1,
                  GLX_TRANSPARENT_TYPE_EXT, GLX_TRANSPARENT_RGB_EXT,
                  None };
```

2. Create a frame and form, then create the window widget, attaching it to the form on all four sides. Add expose, resize, and input callbacks.

```
/* specify visual directly */
if (!(visinfo = glXChooseVisual(dpy, DefaultScreen(dpy), attribs)))
XtAppError(appctx, "no suitable RGB visual");

/* attach to form on all 4 sides */
n = 0;
XtSetArg(args[n], XtNx, 0); n++;
XtSetArg(args[n], XtNy, 0); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], GLwNvisualInfo, visinfo); n++;
state.w = XtCreateManagedWidget("glxwidget",
    glwMDrawingAreaWidgetClass, form, args, n);
XtAddCallback(state.w, GLwNexposeCallback, expose, NULL);
XtAddCallback(state.w, GLwNresizeCallback, resize, &state);
XtAddCallback(state.w, GLwNinputCallback, input, NULL);
state.cx = glXCreateContext(dpy, visinfo, 0, GL_TRUE);
```

3. Create the overlay widget, using the overlay visual attributes specified in Step 1 and attaching it to the same form as the window. This assures that when the window is moved or resized, the overlay is as well.

```
if (!(visinfo = glXChooseVisual(dpy, DefaultScreen(dpy),
    ov_attribs)))
    XtAppError(appctx, "no suitable overlay visual");
XtSetArg(args[n-1], GLwNvisualInfo, visinfo);
ov_state.w = XtCreateManagedWidget("overlay",
    glwMDrawingAreaWidgetClass, form, args, n);
```

4. Add callbacks to the overlay.

```
XtAddCallback(ov_state.w, GLwNexposeCallback, ov_expose, NULL);
XtAddCallback(ov_state.w, GLwNresizeCallback, resize, &ov_state);
XtAddCallback(ov_state.w, GLwNinputCallback, input, NULL);
ov_state.cx = glXCreateContext(dpy, visinfo, 0, GL_TRUE);
```

   Note that the overlay uses the same resize and input callback:

   - For resize, you may or may not wish to share callbacks, depending on the desired functionality; for example, if you have a weathermap with annotations, both should resize in the same fashion.
   - For input, the overlay usually sits on top of the normal window and receives the input events instead of the overlay window. Redirecting both to the same callback guarantees that you receive the events regardless of which window actually received them.
   - The overlay has its own expose function: each time the overlay is exposed, it redraws itself.

5. Call **XRaiseWindow()** to make sure the overlay is on top of the window.

```
XRaiseWindow(dpy, XtWindow(ov_state.w));
```

Overlay Troubleshooting

This section gives some advice on issues that can easily cause problems in a program using overlays:

- **Colormaps**. Overlays have their own colormaps. You therefore should call **XSetWMColormapWindows()** to create the colormap, populate it with colors, and to install it.

Note: Overlays on Silicon Graphics systems reserve pixel zero as the transparent pixel. If you attempt to create the colormap with AllocAll, the **XCreateColormap()** function will fail with a BadAlloc X protocol error. Instead of AllocAll, use AllocNone and allocate all the color cells except zero.

- **Window hierarchy**. Overlay windows are created like other windows, their parent window depends on what you pass in at window creation time. Overlay windows can be part of the same window hierarchy as normal windows and be children of the normal windows. An overlay and its parent window are handled as a single hierarchy for events like clipping, event distribution, and so on.
- **Color limitations**. On low-end Silicon Graphics systems, there are only a few overlay planes available; thus, items drawn in the overlay planes (such as menus) usually use only a few colors—no more than three colors and the transparent pixel in some cases. More recent low-end systems (24-bit Indy™ graphics), mid-range systems (IMPACT™), and high-end systems (RealityEngine™) support 8-bit overlay planes.
- **Input events.** The overlay window normally sits on top of the normal window. Thus, it gets all input events such as mouse and keyboard events. If the application is only waiting for events on the normal window, it will not get any of those events. It is necessary to select for events on the overlay window as well.
- **Not seeing the overlay**. Although overlay planes are conceptually considered to be "above" the normal plane, an overlay window can be below a normal window and thus clipped by it. When creating an overlay and a normal window, use **XRaiseWindow()** to ensure that the overlay window is on top of the normal window. If you use Xt, you must do this after the widget hierarchy has been realized.

Rubber Banding

Rubber banding can be used for cases where it's necessary to draw a few lines over a scene in response to a mouse movement. An example is the movable window outline that you get when resizing or moving a window. Rubber-banding is also used frequently by drawing programs.

The *4Dwm* window manager provides rubber banding for moving and resizing windows. However, if you need rubber banding features inside your application, you have to manage it yourself.

Here's the best way to do rubber banding with overlays (this is the method used by *4Dwm*, the default Silicon Graphics window manager):

1. Map an overlay window, with its *background* pixmap set to None (*background* is passed in as a parameter to **XCreateWindow()**). This window should be as large as the area over which rubber banding could take place.
2. Draw rubber bands in the new overlay window. Ignore resulting damage to other windows in the overlay plane.
3. Unmap the rubber-band window, which sends Expose events to other windows in the overlay plane.

Using Popup Menus With the GLwMDrawingArea Widget

Pop-ups are used by many applications to allow user input. A sample program, *simple-popup.c*, is included in the source tree. It uses the function **XmCreateSimplePopupMenu()** to add a popup to a drawing area widget.

Note that if you are not careful when you create a popup menu as a child of GLwMDrawingArea widget, you may get a BadMatch X protocol error: The menu (like all other Xt shell widgets) inherits its default colormap and depth from the GLwMDrawingArea widget, but its default visual from the parent (root) window. Since the GLwMDrawingArea widget is normally not the default visual, the menu inherits a nondefault depth and colormap from the GLwMDrawingArea widget, but also inherits its visual from the root window (that is, inherits the default visual); leading to a BadMatch X protocol error. See "Inheritance Issues" on page 38 for more detail and for information on finding the error.

There are two ways to work around this:

- Specify the visual, depth, and colormap of the menu explicitly. If you do that, consider putting the menu in the overlay plane.
- Make the menu a child of a widget that is in the default visual; for example, if the GLwMDrawingArea widget is a child of an XmFrame, make the menu a child of XmFrame as well. Example 4-1 provides a code fragment from *motif/simple-popup.c*.

Example 4-1 Popup Code Fragment

```
static void
create_popup(Widget parent) {
    Arg args[10];
    static Widget popup;
    int n;
    XmButtonType button_types[] = {
        XmPUSHBUTTON, XmPUSHBUTTON, XmSEPARATOR, XmPUSHBUTTON, };

    XmString button_labels[XtNumber(button_types)];

    button_labels[0] = XmStringCreateLocalized(“draw filled”);
    button_labels[1] = XmStringCreateLocalized(“draw lines”);
    button_labels[2] = NULL;
    button_labels[3] = XmStringCreateLocalized(“quit”);

    n = 0;
    XtSetArg(args[n], XmNbuttonCount, XtNumber(button_types)); n++;
    XtSetArg(args[n], XmNbuttonType, button_types); n++;
    XtSetArg(args[n], XmNbuttons, button_labels); n++;
    XtSetArg(args[n], XmNsimpleCallback, menu); n++;
    popup = XmCreateSimplePopupMenu(parent, “popup”, args, n);
    XtAddEventHandler(parent, ButtonPressMask, False, activate_menu,
                     &popup);
    XmStringFree(button_labels[0]);
    XmStringFree(button_labels[1]);
    XmStringFree(button_labels[3]);
}
main(int argc, char *argv[]) {
    Display        *dpy;
    XtAppContext    app;
    XVisualInfo    *visinfo;
    GLXContext      glxcontext;
    Widget          toplevel, frame, glxwidget;

    toplevel = XtOpenApplication(&app, “simple-popup”, NULL, 0, &argc,
                  argv, fallbackResources, applicationShellWidgetClass,
                  NULL, 0);
    dpy = XtDisplay(toplevel);

    frame = XmCreateFrame(toplevel, “frame”, NULL, 0);
    XtManageChild(frame);

    /* specify visual directly */
```

```
    if (!(visinfo = glXChooseVisual(dpy, DefaultScreen(dpy), attribs)))
        XtAppError(app, "no suitable RGB visual");

    glxwidget = XtVaCreateManagedWidget("glxwidget",
                    glwMDrawingAreaWidgetClass, frame, GLwNvisualInfo,
                    visinfo, NULL);
    XtAddCallback(glxwidget, GLwNexposeCallback, expose, NULL);
    XtAddCallback(glxwidget, GLwNresizeCallback, resize, NULL);
    XtAddCallback(glxwidget, GLwNinputCallback, input, NULL);

    create_popup(frame);

    XtRealizeWidget(toplevel);

    glxcontext = glXCreateContext(dpy, visinfo, 0, GL_TRUE);
    GLwDrawingAreaMakeCurrent(glxwidget, glxcontext);

    XtAppMainLoop(app);
}
```

Using Visuals

This section explains how to choose and use visuals on Silicon Graphics workstations. It discusses the following topics:

- "Some Background on Visuals"
- "Running OpenGL Applications Using a Single Visual"

Some Background on Visuals

An X visual defines how pixels in a window are mapped to colors on the screen. Each window has an associated visual, which determines how pixels within the window are displayed on screen. GLX overloads X visuals with additional framebuffer capabilities needed by OpenGL.

Table 4-1 illustrates which X visuals support which type of OpenGL rendering, and whether the colormaps for those visuals are writable or not. Visuals that aren't available on Silicon Graphics systems are marked with an asterisk.

Table 4-1 X Visuals and Supported OpenGL Rendering Modes

OpenGL Rendering Mode	X visual	Writable colormap?
RGBA	TrueColor	no
RGBA	DirectColor*	yes
color index	PseudoColor	yes
color index	StaticColor*	no
not supported	GrayScale	yes
not supported	StaticGray	no

* Not supported on Silicon Graphics systems.

An X server can provide multiple visuals, depending on the available hardware and software support. Each server has a default visual that can be specified when the server starts. You can determine the default visual with the Xlib macro **DefaultVisual()**.

Because you can't predict the configuration of every X server, and you may not know the system configuration your program will be used on, it is best to find out what visual classes are available on a case-by-case basis.

- From the command line, execute *xdpyinfo* for a list of all visuals the server supports.
- Execute *glxinfo* or *findvis* to find visuals that are capable of OpenGL rendering. The *findvis* command can actually look for available visuals with certain attributes. See the reference page for more information.
- From within your application, use the Xlib functions **XGetVisualInfo()** and **XMatchVisualInfo()**—or **glXGetConfig()**—or the GLX function **glXChooseVisual()**.

Note: For most applications, using OpenGL RGBA color mode and a TrueColor visual is recommended.

Running OpenGL Applications Using a Single Visual

Note: This section applies only to IRIS IM.

In previous chapters, this guide has assumed separate visuals for the X and OpenGL portions of the program. The top-level windows and all parts of the application that aren't written in OpenGL use the default visual (typically 8-bit PseudoColor, but it depends on the configuration of the server). OpenGL runs in a single window that uses an Open GL visual.

An alternative approach is to run the whole application using an OpenGL visual. To do this, determine the suitable OpenGL visual (and colormap and pixel depth) at the start of the program and create the top-level window using that visual (and colormap and pixel depth). Other windows, including the OpenGL window, inherit the visual. When you use this approach, there is no need to use the GLwMDrawingArea widget; the standard IRIS IM XmDrawingArea works just as well.

The advantages of using a single visual include the following:

- **Simplicity**. Everything uses the same visual, so you don't have to worry about things like colormap installation more than once in the application. (However, if you use the GLwMDrawingArea widget, it does colormap installation for you—see "Drawing-Area Widget Setup and Creation" on page 29.)
- **Reduced colormap flashing**. Colormap flashing happens if several applications are running, each using its own colormap, and you exceed the system's capacity for installed hardware colormaps. Flashing is reduced for a single visual because the entire application uses a single colormap. The application can still cause other applications to flash, but all recent Silicon Graphics systems have multiple hardware colormaps to reduce flashing.
- **Easier mixing of OpenGL and X**. If you run in a single visual, you can render OpenGL to any window in the application, not just to a dedicated window. For example, you could create an XmDrawnButton and render OpenGL into it.

The advantages of using separate visuals for X and OpenGL include the following:

- **Consistent colors in the X visual**. If the OpenGL visual has a limited number of colors, you may want to allow more colors for X. For example, if you're using double buffering on an 8-bit machine, you have only 4 bitplanes (16 colors) per buffer. You can have OpenGL dither in such a circumstance to obtain approximations of other colors, but X won't dither, so if you're using the same visual for OpenGL and X, the X portion of your application will be limited to 16 colors as well.

This limiting of colors would be particularly unfortunate if your program uses the Silicon Graphics color-scheme system. While X chooses a color as close as possible to the requested color, the choice is usually noticeably different from the requested color. As a result, your application looks noticeably different from the other applications on the screen.

- **Memory savings**. The amount of memory used by a pixmap within the X server depends on the depth of the associated visual. Most applications use X pixmaps for shadows, pictures, and so on that are part of the user interface widgets. If you're using a 12-bit or 24-bit visual for OpenGL rendering and your program also uses X pixmaps, those pixmaps would use less memory in the default 8-bit visual than in the OpenGL visual.
- **Easier menu handling in IRIS IM**. If the top-level shell is not in the default visual, there will be inheritance problems during menu creation (see "Inheritance Issues" on page 38). You have to explicitly specify the visual depth and colormap when creating a menu. For cascading menus, specify depth and colormap separately for each pane.

Using Colormaps

This section explains using colormaps in some detail. Note that in many cases, you won't need to worry about colormaps: Just use the drawing area widget and create a TrueColor visual for your RGBA OpenGL program. However, under certain circumstances, for example, if the OpenGL program uses indexed color, the information in this section is important. The section discusses these topics:

- "Background Information About Colormaps"
- "Choosing Which Colormap to Use"
- "Colormap Example"

Background Information About Colormaps

OpenGL supports two rendering modes: RGBA mode and color index mode.

- In RGBA mode, color buffers store red, green, blue, and alpha components directly.
- In color-index mode, color buffers store indexes (names) of colors that are dereferenced by the display hardware. A color index represents a color by name rather than value. A colormap is a table of index-to-RGB mappings.

OpenGL color modes are discussed in some detail in the section "RGBA versus Color-Index Mode" in Chapter 5, "Color," of the *OpenGL Programming Guide.*

The X Window System supports six different types of visuals, with each type using a different type of colormap (see Table 4-1 on page 67). Although working with X colormaps may initially seem somewhat complicated, the X Window System does allow you a great deal of flexibility in choosing and allocating colormaps. Colormaps are discussed in detail and with example programs in Chapter 7, "Color," of O'Reilly Volume One.

The rest of this section addresses some issues having to do with X colormaps.

Color Variation Across Colormaps

The same index in different X colormaps does not necessarily represent the same color. Be sure you use the correct color index values for the colormap you are working with.

If you use a nondefault colormap, avoid color macros such as **BlackPixel()** and **WhitePixel()**. As is required by X11, these macros return pixel values that are correct for the default colormap but inappropriate for your application. The pixel value returned by the macro is likely to represent a color different from black or white in your colormap, or worse yet, be out of range for it. If the pixel value does not exist in your colormap (such as any pixel greater than three for a 2-bit overlay colormap), an X protocol error results.

A "right index–wrong map" type of mistake is most likely if you use the macros **BlackPixel** and **WhitePixel**. For example, the **BlackPixel** macro returns zero, which is black in the default colormap. That value is always transparent (not black) in a popup or overlay colormap (if it supports transparent pixels).

You might also experience problems with colors not appearing correctly on the screen because the colormap for your window is not installed in the hardware.

Multiple Colormap Issues

The need to deal with multiple colormaps of various sizes raises new issues. Some of these issues do not have well-defined solutions.

There is no default colormap for any visual other than the default visual. You must tell the window manager which colormaps to install using **XSetWMColormapWindows()**, unless you use the GLwMDrawingArea widget, which does this for you.

- With multiple colormaps in use, colormap flashing may occur if you exceed the hardware colormap resources.
- An application has as many of its colormaps installed as possible only when it has colormap focus.
  - At that time, the window manager attempts to install all the application's colormaps, regardless of whether or not all are currently needed. These colormaps remain installed until another application needs to have one of them replaced.
  - If another application gets colormap focus, the window manager installs that application's (possibly conflicting) colormaps. Some widgets may be affected while other widgets remain unchanged.
  - The window manager does not reinstall the colormaps for your application until your application has the colormap focus again.

The **getColormap()** call defined in Example 4-2 returns a sharable colormap (the ICCCM RGB_DEFAULT_MAP) for a TrueColor visual given a pointer to XVisualInfo. This is useful to reduce colormap flashing for non-default visuals.

Example 4-2 Retrieving the Default Colormap for a Visual

```
Colormap
getColormap(XVisualInfo * vi)
{
    Status           status;
    XStandardColormap *standardCmaps;
    Colormap         cmap;
    int              i, numCmaps;

    /* be lazy; using DirectColor too involved for this example */
    if (vi->class != TrueColor)
        fatalError("no support for non-TrueColor visual");
    /* if no standard colormap but TrueColor, make an unshared one */
```

```
    status = XmuLookupStandardColormap(dpy, vi->screen, vi->visualid,
        vi->depth, XA_RGB_DEFAULT_MAP,
        /* replace */ False, /* retain */ True);
    if (status == 1) {
        status = XGetRGBColormaps(dpy, RootWindow(dpy, vi->screen),
                                 &standardCmaps, &numCmaps,
                                 XA_RGB_DEFAULT_MAP);
        if (status == 1)
            for (i = 0; i < numCmaps; i++)
                if (standardCmaps[i].visualid == vi->visualid) {
                    cmap = standardCmaps[i].colormap;
                    XFree(standardCmaps);
                    return cmap;
                }
    }
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
        vi->visual, AllocNone);
    return cmap;
}
```

Choosing Which Colormap to Use

When choosing which colormap to use, follow these heuristics:

1. First decide whether your program will use RGBA or color-index mode. Some operations, such as texturing and blending, are not supported in color index mode; others, such as lighting, work differently in the two modes. Because of that, RGBA rendering is usually the right choice. (See "Choosing between RGBA and Color-Index Mode" in Chapter 5, "Color," of the *OpenGL Programming Guide*).

   OpenGL 1.0 and 1.1 and GLX 1.0, 1.1, and 1.2 require an RGBA mode program to use a TrueColor or DirectColor visual, and require a color index mode program to use a PseudoColor or StaticColor visual.

   Note: Remember that RGBA is usually the right choice for OpenGL on a Silicon Graphics system.

2. Choose a visual. If you intend to use RGBA mode, specify RGBA in the attribute list when calling **glXChooseVisual()**.

   If RGBA is not specified in the attribute list, **glXChooseVisual()** selects a PseudoColor visual to support color index mode (or a StaticColor visual if no PseudoColor visual is available).

If the framebuffer configuration extension is available, you can use a TrueColor or DirectColor visual in color index mode. See “The Framebuffer Configuration Extension” on page 171.

3. Create a colormap that can be used with the selected visual.
4. If a PseudoColor or DirectColor visual has been selected, initialize the colors in the colormap.

   Note: DirectColor visuals are not supported on Silicon Graphics systems. Colormaps for TrueColor and StaticColor visuals are not writable.
5. Make sure the colormap is installed. Depending on what approach you use, you may or may not have to install it yourself:
   - If you use the GLwMDrawingArea widget, the widget automatically calls **XSetWMColormapWindows()** when the GLwNinstallColormap resource is enabled.
   - The colormap of the top-level window is used if your whole application uses a single colormap. In that case, you have to make sure the colormap of the top level window supports OpenGL.
   - Call **XSetWMColormapWindows()** to ensure that the window manager knows about your window’s colormap. Here’s the function prototype for **XSetWMColormapWindows()**:

     ```
     Status XSetWMColormapWindows(Display *display, Window w,
                                  Window *colormap_windows, int count)
     ```

Many OpenGL applications use a 24-bit TrueColor visual (by specifying GLX_RGBA in the visual attribute list when choosing a visual). Colors usually look right in TrueColor, and some overhead is saved by not having to look up values in a table. On some systems, using 24-bit color can slow down the frame rate since more bits must be updated per pixel, but this isn’t usually a problem.

If you want to adjust or rearrange values in a colormap, you may have to use a PseudoColor visual, which has to be used with color-index mode unless the framebuffer configuration extension is available. Lighting and antialiasing are difficult in color-index mode, and texturing and accumulation don’t work at all. It may be easier to use double-buffering and redraw to produce a new differently-colored image, or use the overlay plane. In general, avoid using PseudoColor visuals if possible.

Overlays, which always have PseudoColor colormaps on current systems, are an exception to this.

Colormap Example

Here's a brief example that demonstrates how to store colors into a given colormap cell:

```
XColor xc;
display = XOpenDisplay(0);
visual = glXChooseVisual(display, DefaultScreen(display),
                         attributeList);
context = glXCreateContext (display, visual, 0, GL_FALSE);
colorMap = XCreateColormap (display, RootWindow(display,
    visual->screen), visual->visual, AllocAll);
    ...
if (ind < visual->colormap_size) {
    xc.pixel = ind;
    xc.red = (unsigned short)(red * 65535.0 + 0.5);
    xc.green = (unsigned short)(green * 65535.0 + 0.5);
    xc.blue = (unsigned short)(blue * 65535.0 + 0.5);
    xc.flags = DoRed | DoGreen | DoBlue;
    XStoreColor (display, colorMap, &xc);
}
```

Caution: Do not use AllocAll on overlay visuals with transparency. If you do, **XCreateColormap()** fails because the transparent cell is read-only.

Stereo Rendering

Silicon Graphics systems and OpenGL both support stereo rendering. In stereo rendering, the program displays a scene from two slightly different viewpoints to simulate stereoscopic vision, resulting in a 3D image to a user wearing a special viewing device. Various viewing devices exist; most of them cover one eye while the computer displays the image for the other eye, then cover the second eye while the computer displays the image for the first eye.

Note: Be sure to look at the *stereo* reference page for more information on stereo rendering (including sample code fragments and pointers to sample code).

In this section, you learn about

- "Stereo Rendering Background Information"
- "How to Do Stereo Rendering"

Stereo Rendering Background Information

There are two basic approaches to stereo rendering, “Quad Buffer Stereo” and “Divided-Screen Stereo.”

Quad Buffer Stereo

Quad buffer stereo uses a separate buffer for the left and right eye, resulting in four buffers if the program is already using a front and back buffer for animation. Quad buffer stereo is supported on RealityEngine and Maximum IMPACT and will be supported on future high-end systems.

The main drawback of this approach is that it needs a substantial amount of framebuffer resources and is therefore feasible only on high-end systems. See “Performing Stereo Rendering on High-End Systems” on page 76 for step-by-step instructions.

Divided-Screen Stereo

Divided-screen stereo divides the screen into left and right pixel lines. This approach is usually appropriate on low-end systems, which don’t have enough memory for quad-buffer stereo.

If you put the monitor in stereo mode, you lose half of the screen’s vertical resolution and pixels get a 1 x 2 aspect ratio. The XSGIvc extension does all X rendering in both parts of the screen. Note, however, that monoscopic OpenGL programs will look wrong if you use the extension.

When working with divided-screen stereo, keep in mind the following caveats:

- Since stereo is enabled and disabled without restarting the server, the advertised screen height is actually twice the height displayed.
- With quad-buffering, stereo pixels are square. If you’re using divided-screen stereo, pixels are twice as high as they are wide. Thus, transformed primitives and images need an additional correction for pixel aspect ratio.

For More Information

See the reference pages for the following functions: XSGIStereoQueryExtension(), XSGIStereoQueryVersion(), XSGIQueryStereoMode(), XSGISetStereoMode(), XSGISetStereoBuffer().

How to Do Stereo Rendering

This section first explains how to do stereo rendering on high-end systems, then on low-end and mid-range systems.

Performing Stereo Rendering on High-End Systems

To perform stereo rendering on high-end systems (RealityEngine, Maximum IMPACT, and future high-end systems), follow these steps:

1. Perform initialization, that is, make sure the GLX extension is supported and so on.
2. Put the monitor in stereo mode with the *setmon* command.
3. Choose a visual with front left, front right, back left, and back right buffers.
4. Perform all other setup operations illustrated in the examples in the previous two chapters: create a window, create a context, make the context current, and so on.
5. Start the event loop.
6. Draw the stereo image:

```
glDrawBuffer(GL_BACK_LEFT);
< draw left image >
glDrawBuffer(GL_BACK_RIGHT);
< draw right image >
glXSwapBuffers(...);
```

For more information, see the **glDrawBuffer()** reference page.

Performing Stereo Rendering on Low-End and Mid-Range Systems

To perform stereo rendering on low-end and mid-range systems (including High IMPACT), follow these steps:

1. Perform initialization, that is, make sure the GLX extension is supported and so on.
2. Put the monitor in stereo mode using the *setmon* command.
3. Call **XSGIStereoQueryExtension()** to see if the stereo extension is supported.
   - If stereo is not supported, exit.
   - If stereo is supported, call **XSGISetStereoMode()** to turn it on (options are STEREO_BOTTOM or STEREO_TOP).

4. Choose visual with front left, front right, back left and back right buffers by calling **glXChooseVisual** with both GLX_DOUBLEBUFFER and GLX_STEREO in the attribute list.
5. Perform all other setup operations discussed in the examples in the previous two chapters: create a window, create a context, make the context current, and so on.
6. To draw the stereo image, use code similar to this pseudo-code fragment:

```
XSGISetStereoBuffer(STEREO_BUFFER_LEFT);
< draw left image >
XSGISetStereoBuffer(STEREO_BUFFER_RIGHT);
< draw right image >
glXSwapBuffers(...);
```

Using Pixmaps

An OpenGL program can render to two kinds of drawables: windows and pixmaps. (Rendering to PBuffers is also possible if that extension is supported. See “The Pixel Buffer Extension” on page 179.) A pixmap is an offscreen rendering area. On Silicon Graphics systems, pixmap rendering is not hardware accelerated.

In contrast to windows, where drawing has no effect if the window is not visible, a pixmap can be drawn to at any time since it resides in memory. Before the pixels in the pixmap become visible, they have to be copied into a visible window. The unaccelerated rendering for pixmap pixels has performance penalties.

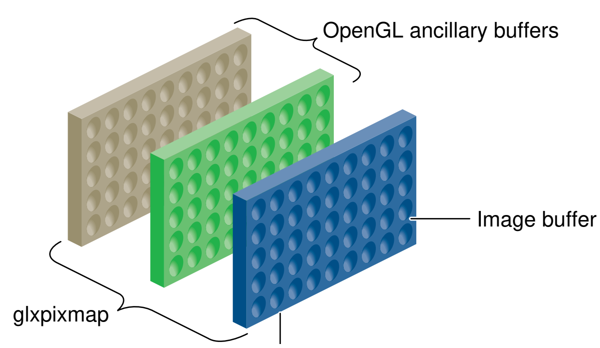


Figure 4-2 X Pixmaps and GLX Pixmaps

This section explains how to create and use a pixmap and looks at some related issues:

- "Creating and Using Pixmaps" provides basic information about working with pixmaps.
- "Direct and Indirect Rendering" provides some background information; it is included here because rendering to pixmaps is always indirect.

Creating and Using Pixmaps

Integrating an OpenGL program with a pixmap is very similar to integrating it with a window. It involves the steps given below. (Note that Steps 1-3 and Step 6 are discussed in detail in "Integrating Your OpenGL Program With IRIS IM" on page 15.)

1. Open the connection to the X server.
2. Choose a visual.
3. Create a rendering context with the chosen visual.

   This context must be indirect.
4. Create an X pixmap using **XCreatePixmap().**
5. Create a GLX pixmap using **glXCreateGLXPixmap().**

   ```
   GLXPixmap glXCreateGLXPixmap(Display *dpy, XVisualInfo *vis,
                                 Pixmap pixmap)
   ```

   The GLX pixmap "wraps" the pixmap with ancillary buffers determined by *vis* (see Figure 4-2).

   The *pixmap* parameter must specify a pixmap that has the same depth as the visual that *vis* points to (as indicated by the visual's GLX_BUFFER_SIZE value), or a BadMatch X protocol error results.
6. Use **glXMakeCurrent()** to bind the pixmap to the context.

   You can now render into the GLX pixmap.

Direct and Indirect Rendering

OpenGL rendering is done differently in different rendering contexts (and on different platforms).

- **Direct rendering** contexts support rendering directly from OpenGL via the hardware, bypassing X entirely. Direct rendering is much faster than indirect rendering, and all Silicon Graphics systems can do direct rendering to a window.
- In **indirect rendering** contexts, OpenGL calls are passed by GLX protocol to the X server, which does the actual rendering. Remote rendering has to be done indirectly; pixmap rendering is implemented to work only indirectly.

Note: As a rule, use direct rendering unless you're using pixmaps. If you ask for direct and your DISPLAY is remote, the library automatically switches to indirect rendering.

Here's some more information about indirect rendering:

In indirect rendering, OpenGL rendering commands are added to the GLX protocol stream, which in turn is part of the X protocol stream. Commands are encoded and sent to the X server. Upon receiving the commands, the X server decodes them and dispatches them to the GLX extension. Control is then given to the GLX process (via a context switch) so the rendering commands can be processed. The faster the graphics hardware, the higher the overhead from indirect rendering.

You can obtain maximum indirect-rendering speed by using display lists; they require a minimum of interaction with the X server. Unfortunately, not all applications can take full advantage of display lists; this is particularly a problem in applications using rapidly-changing scene structures. Display lists are efficient because they reside in the X server.

You may see multiple XSGI processes on your workstation when you are running indirect rendering OpenGL programs.

Performance Considerations for X and OpenGL

Due to synchronization and context switching overhead, there is a possible performance hit for mixing OpenGL and X in the same window. GLX does not constrain the order in which OpenGL commands and X requests are executed. To ensure a particular order, use the GLX commands **glXWaitGL()** and **glXWaitX()**.

- **glXWaitGL()** prevents any subsequent X calls from executing until all pending OpenGL calls complete. When you use indirect rendering, this function does not contact the X server and is therefore more efficient than **glFinish().**

- **glXWaitX()**, when used with indirect rendering, is just the opposite: it makes sure that all pending X calls complete before any further OpenGL calls are made. This function, too, doesn't need to contact the X server, giving it an advantage over **XSync()** when rendering indirectly.
- Remember also to batch Expose events. See "Exposing a Window" on page 47.
- Make sure no additional Expose events are already queued after the current one. You can discard all but the last event.

Portability

If you expect to port your program from X to other windowing systems (such as Windows NT), certain programming practices make porting easier. Here's a partial list:

- Isolate your windowing functions and calls from your rendering functions. The more modular your code is in this respect, the easier it is to switch to another windowing system.
- (For Windows NT porting only) Avoid naming variables with any variation of the words "near" and "far"—they are reserved words in Intel™ xx86 compilers. For instance, you should avoid the names _near, _far, __near, __far, near, far, Near, Far, NEAR, FAR, and so on.
- Windows NT programs by default have a small stack; don't allocate large arrays on the stack.
- Windows NT does not have an equivalent to **glXCopyContext().**

Chapter 5

Introduction to OpenGL Extensions

OpenGL extensions introduce new features and enhance performance. Some extensions provide completely new functionality; for example, the convolution extension allows you to blur or sharpen images using a filter kernel. Other extensions enhance existing functionality; for example, the subtexture extension lets you replace a portion of a texture without reloading the entire image.

Several extensions provide IRIS GL functionality that is not available in OpenGL 1.0. If you are porting a program from IRIS GL to OpenGL, you may therefore find some extensions particularly helpful. See Appendix A, "OpenGL and IRIS GL," for a list of IRIS GL commands and corresponding OpenGL functionality.

This chapter provides basic information about OpenGL extensions. You learn about:

- "Determining Extension Availability" on page 81
- "Finding Information About Extensions" on page 84

Determining Extension Availability

Function names and tokens for OpenGL extensions have EXT or a vendor-specific acronym as a suffix, for example **glConvolutionFilter2DEXT()** or **glColorTableSGI()**. The names of the extensions themselves (the extension strings) use prefixes, for example, SGI_color_table. Here's a detailed list of all suffixes and prefixes:

- EXT is used for extensions that have been reviewed and approved by more than one OpenGL vendor. (Typically these are supported by at least two OpenGL vendors.)
- SGI is used for extensions that may be found across the Silicon Graphics product line, although the support for all products may not appear in the same release.
- SGIS is used for extensions that can be found only on a subset of Silicon Graphics platforms.
- SGIX is used for extensions that are experimental: in future releases, the API for these extensions may change, or they may not be supported at all.

How to Check for OpenGL Extension Availability

All supported extensions have a corresponding definition in *gl.h* and a token in the extensions string returned by **glGetString()**. For example, if the ABGR extension (EXT_abgr) is supported, it is defined in *gl.h* as follows:

```
#define GL_EXT_abgr 1
```

GL_EXT_abgr appears in the extensions string returned by **glGetString()**. Use the definitions in *gl.h* at compile time to determine if procedure calls corresponding to an extension exist in the library.

Applications should do compile-time checking—for example, making sure GL_EXT_abgr is defined; and run-time checking—for example, making sure GL_EXT_abgr is in the extension string returned by **glGetString()**.

- Compile-time checking ensures that entry points such as new functions or new enums are supported. You can't compile or link a program that uses a certain extension if the client side development environment does not support it.
- Run-time checking ensures that the extension is supported for the OpenGL server and run-time library you are using.

  Note that availability depends not only on the operating system but also on the particular hardware you are using: even though the 5.3 OpenGL library supports GL_CONVOLUTION_2D_EXT, you get an GL_INVALID_OPERATION error if you call **glConvolutionFilter2DEXT()** on an Indy system.

Example Program: Checking for Extension Availability

In Example 5-1, the function **QueryExtension()** checks whether an extension is available.

Example 5-1 Checking for Extensions

```
main(int argc, char* argv[]) {
...
    if (!QueryExtension("GL_EXT_texture_object")) {
        fprintf(stderr, "texture_object extension not supported.\n");
        exit(1);
    }
...
}
```

```
static GLboolean QueryExtension(char *extName)
{
    /*
    ** Search for extName in the extensions string. Use of strstr()
    ** is not sufficient because extension names can be prefixes of
    ** other extension names. Could use strtok() but the constant
    ** string returned by glGetString might be in read-only memory.
    */
    char *p;
    char *end;
    int extNameLen;

    extNameLen = strlen(extName);

    p = (char *)glGetString(GL_EXTENSIONS);
    if (NULL == p) {
        return GL_FALSE;
    }

    end = p + strlen(p);

    while (p < end) {
        int n = strcspn(p, " ");
        if ((extNameLen == n) && (strncmp(extName, p, n) == 0)) {
            return GL_TRUE;
        }
        p += (n + 1);
    }
    return GL_FALSE;
}
```

As an alternative to checking for each extension explicitly, you can make the following calls to determine the system and IRIX release on which your program is running:

```
glGetString(GL_RENDERER)
...
glGetString(GL_VERSION)
```

Given a list of extensions supported on that system for that release, you can determine whether the particular extension you need is available. For this to work on all systems, a table of different systems and the extensions supported has to be available.

When an extension is incomplete, it is not advertised in the extensions string. Some of the RealityEngine extensions supported in IRIX 5.3 (for example, the subtexture, sharpen texture, convolution, and histogram extensions) fall in that category.

Checking for GLX Extension Availability

If you use any of the extensions to GLX, described in Chapter 9, "Extensions to GLX," you also need to check for GLX extension availability.

Querying for GLX extension support is similar to querying for OpenGL extension support with the following exceptions:

- Compile time defines are found in *glx.h.*
- To get the list of supported GLX extensions, call **glXQueryExtensionsString().**
- GLX versions must be 1.1 or greater (no extensions to GLX 1.0 exist).

Adapt the process described in "How to Check for OpenGL Extension Availability" on page 82 taking these exceptions into account.

Finding Information About Extensions

While this book provides rather detailed coverage of several extensions, you may find it helpful to look at the *glintro* reference page for information about the current state of extensions on your system. See the *glXintro* reference page for more information about GLX extensions on your system.

Note that reference pages for extensions are included where the extension is available. Other reference pages that have new options available because of extensions have been updated accordingly.

All complete example programs (though not the short code fragments) are available in */usr/share/src/OpenGL* if you have the *ogl_dev.sw.samples* subsystem installed.

Chapter 6

Texturing Extensions

This chapter explains how to use the different OpenGL texturing extensions. You learn about:

The Texture Extension

The texture extension, EXT_texture, allows you to select an internal format for textures. When you create a texture using OpenGL 1.0, you can only define it as having 1, 2, 3, or 4 components. The texture extension provides several additional predefined formats. You can indicate the following on a per-texture basis:

- the number of components to store in texture memory
- the internal image format (GL_RGB4_EXT and so on, see Table 6-1)

The extension also defines a robust method for applications to determine what combinations of texture dimensions and resolutions are supported. It also introduces a new texture environment: GL_REPLACE_EXT.

This extension is the foundation for other extensions discussed in this chapter.

Note: Some of the functionality of this extension was available in IRIS GL on RealityEngine.

Why Use the Texture Extension?

Using the texture extension, you can improve your program as follows:

- **Save space**. Use the formats best suited for the texture-memory size of a specific system. Smaller internal formats use less texture memory so more textures can be simultaneously resident in texture memory. This can be especially useful if you want to fit a texture into the 16-bit texture storage on current machines.
- **Increase speed**. The extension offers a fine-grained selection of sizes. At times, you may prefer using 16-bit textures to using 32-bit textures because the increase in rendering speed outweighs the slight decrease in image quality. Smaller textures are usually faster and may therefore be preferable, if sufficient.
- **Predict texture size**. Use the texture extension's proxy mechanism to determine whether a texture you intend to use fits into available texture memory.

Using the Texture Extension

To use the texture extension, call **glTexImage1D()** or **glTexImage2D()**, specifying one of the tokens defined by this extension as the *components* parameter.

Table 6-1 lists some of the formats defined by the extension; for a complete list, see the reference page for **glTexImage*()**. The table shows the token names with their corresponding base formats and their desired component resolutions. It also shows Red (R), Green (G), Blue (B), Luminance (L), Alpha (A), and Intensity (I) values for each format. You can query the actual resolution with **glGetTexLevelParameteriv()** using the appropriate token, such as GL_TEXTURE_RED_SIZE_EXT. All OpenGL implementations accept all tokens but may not always allocate a resolution exactly equal to that indicated in the table.

Table 6-1 Some Internal Formats Supported by the Texture Extension

Token Name	Base Format	R	G	B	A	L	I
GL_LUMINANCE8_EXT	GL_LUMINANCE					8	
GL_LUMINANCE4_ALPHA4_EXT	GL_LUMINANCE_ALPHA				4	4	
GL_LUMINANCE12_ALPHA4_EXT	GL_LUMINANCE_ALPHA				4	12	
GL_LUMINANCE16_ALPHA16_EXT	GL_LUMINANCE_ALPHA				16	16	
GL_INTENSITY8_EXT	GL_INTENSITY_EXT						8
GL_RGB4_EXT	GL_RGB	4	4	4			
GL_RGB8_EXT	GL_RGB	8	8	8			
GL_RGBA4_EXT	GL_RGBA	4	4	4	4		
GL_RGB5_A1_EXT	GL_RGBA	5	5	5	1		
GL_RGBA8_EXT	GL_RGBA	8	8	8	8		

The extension redefines the application of a texture to the color components of a fragment. In particular, a new texture environment, GL_REPLACE_EXT, and two new texture formats, GL_ALPHA and GL_INTENSITY_EXT, are available. Table 6-2 illustrates this, with the abbreviations having the following meanings:

Rv, Gv, Bv, Av	result of the texture environment function
Rt, Gt, Bt, At	texture color
Rf, Gf, Bf, Af	fragment color
Rc, Gc, Bc, Ac	texture environment color (see GL_TEXTURE_ENV_COLOR)

Table 6-2 Texture Functions

Base Texture Format	Replace	Modulate	Blend	Decal
GL_ LUMINANCE	Rv = Lt Gv = Lt Bv = Lt Av = Af	Rv = Rf * Lt Gv = Gf * Lt Bv = Bf * Lt Av = Af	Rv = Rf * (1-Lt) + Rc* Lt Gv = Gf * (1-Lt) + Gc * Lt Bv = Bf * (1-Lt) + Bc * Lt Av = Af	undefined
GL_ALPHA	Rv = Rf Gv = Gf Bv = Gf Av = At	Rv = Rf Gv = Gf Bv = Gf Av = Af * At	Rv = Rf Gv = Gf Bv = Bf Av = Af * At	undefined
GL_ INTENSITY_ EXT	Rv = It Gv = It Bv = It Av = It	Rv = Rf * It Gv = Gf * It Bv = Bf * It Av = Af * It	Rv = Rf * (1-It) + Rc * It Gv = Gf * (1-It) + Gc * It Bv = Bf * (1-It) + Bc * It Av = Af * (1-It) + AcIt	undefined
GL_ LUMINANCE _ALPHA	Rv = Lt Gv = Lt Bv = Lt Av = At	Rv = Rf * Lt Gv = Gf * Lt Bv = Bf * Lt Av = Af * At	Rv = Rf * (1-Lt) + Rc * Lt Gv = Gf * (1-Lt) + Gc * Lt Bv = Bf * (1-Lt) + Bc * Lt Av = Af * At	undefined
GL_RGB	Rv = Rt Gv = Gt Bv = Bt Av = Af	Rv = Rf * Rt Gv = Gf * Gt Bv = Bf * Bt Av = Af	Rv = Rf * (1-Rt) + Rc * Rt Gv = Gf * (1-Gt) + Gc * Gt Bv = Bf * (1-Bt) + Bc * Bt Av = Af	Rv = Rt Gv = Gt Bv = Bt Av = Af
GL_RBGA	Rv = Rt Gv = Gt Bv = Bt Av = At	Rv = Rf * Rt Gv = Gf * Gt Bv = Bf * Bt Av = Af * At	Rv = Rf * (1-Rt) + Rc * Rt Gv = Gf * (1-Gt) + Gc * Gt Bv = Bf * (1-Bt) + Bc * Bt Av = Af * At	Rv = Rf * (1-At) + Rt * At Gv = Gf * (1-At) + Gt * At Bv = Bf * (1-At) + Bt * At Av = Af

The Texture Extension Proxy Mechanism

In OpenGL 1.0, it's difficult to find out whether a texture fits into texture memory on a certain system. When you call **glGetIntegerv()** with the parameter GL_MAX_TEXTURE_SIZE, you can't be sure of an accurate result: Texel format, component resolution, and the shape of a texture determine whether the texture fits. To ensure that a texture of any format and resolution fits, the function returns a result based on the worst-case scenario.

A solution to this problem is the proxy mechanism offered by the texture extension. To test whether a certain texture fits, follow these steps:

1. Call **glTexImage2D()** or **glTexImage1D()** with target set to GL_PROXY_TEXTURE_2D_EXT or GL_PROXY_TEXTURE_1D_EXT (see Example 6-1).
2. Test the proxy texture state values to see if the texture would fit and how it would be stored in texture memory by calling **glGetTexLevelParameteriv()** or **glGetTexLevelParameterfv()**. Set *target* to one of the proxy textures: GL_PROXY_TEXTURE_2D_EXT or GL_PROXY_TEXTURE_1D_EXT. Set *pname* to the parameter you are interested in, for example, GL_TEXTURE_WIDTH. If the call returns non-zero, the texture will fit.

   If the texture is too large, all of the proxy state variables are set to zero. If the texture can be accommodated, these values are set to the requested parameters. The proxy 1-D texture behaves like the proxy 2-D texture; however, its state does not include GL_TEXTURE_HEIGHT.

   You still need to call **glTexImage2D()** with *target* GL_TEXTURE_2D to define the actual texture.
3. To determine the maximum array size for a mipmap texture, specify and query the proxy texture at the highest level that accurately reflects the aspect ratio of the desired level zero array.

Example 6-1 provides a code fragment that determines the largest texture that can be allocated for a given internal image format and aspect ratio. The program incrementally selects larger textures to find the maximum that will fit.

Example 6-1 Using the Proxy Mechanism

```
/*
** Demonstrates use of the proxy texture target to probe texture space
** to determine the largest texture which can be allocated for a given
** internal image format and aspect ratio.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <GL/gl.h>
#include <GL/glu.h>

/* (int)floor(log2(x)) */
static int
intFloorLog2(unsigned int x)
{
    int a = 0;
    while (x >>= 1) ++a;
    return a;
}

/* true if x is a power of two */
static GLboolean
isPow2(unsigned int x)
{
    return ((x > 0) && (x & (x - 1) == 0));
}

/* find largest texture with specified internal format and aspect */
static void
findMaxTexture(GLenum internalFormat, int dim,
    int xAspect, int yAspect, int border, int maxMipmapLevel,
    int *widthOut, int *heightOut)
{
    int level = 0, levelOffset = 0;
    int width = 1, height = 1;
    int maxLevel = 0, maxWidth = 0, maxHeight = 0;

    if (xAspect > yAspect) {
        width = xAspect / yAspect;
        levelOffset = intFloorLog2(width);
    } else {
        height = yAspect / xAspect;
        levelOffset = intFloorLog2(height);
    }
```

```
while (1) {
    GLint proxyComponents;

    switch (dim) {
      case 1:
        glTexImage1D(GL_PROXY_TEXTURE_1D_EXT,
                     level, internalFormat,
                     width+2*border, border,
                     GL_RGBA, GL_UNSIGNED_BYTE, NULL);

        glGetTexLevelParameteriv(GL_PROXY_TEXTURE_1D_EXT, level,
                GL_TEXTURE_COMPONENTS, &proxyComponents);
        break;
      case 2:
        glTexImage2D(GL_PROXY_TEXTURE_2D_EXT,
                 level, internalFormat,
                 width+2*border, height+2*border, border,
                 GL_RGBA, GL_UNSIGNED_BYTE, NULL);

        glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D_EXT,level
                 GL_TEXTURE_COMPONENTS,&proxyComponents);
        break;
      default:
        *widthout = 0;
        *heightout = 0;
        return;
    }

    if (proxyComponents != internalFormat) {
       /* proxy allocation failed -- we're done */
        break;
    } else {
       /* proxy allocation succeeded -- see how we did */
           if (level>maxLevel || width>maxWidth ||
                                             height>maxHeight)
        {
            maxLevel = level;
            maxWidth = width << level;
            maxHeight = height << level;
        }
    }

    if (maxMipmapLevel == 0) {
   /* try the next larger image size at level zero */
```

```
                width <<= 1;
                height <<= 1;
            } else {
           /* try same image size at next higher mipmap level */
                ++level;
                if (level+levelOffset > maxMipmapLevel) {
           /* can't query levels which don't exist--we're done */
                    break;
                }
            }
        }

        if (maxWidth>0 && maxHeight>0) {
            maxWidth += 2*border;
            maxHeight += 2*border;
        }
        *widthOut = maxWidth;
        *heightOut = maxHeight;
    }
    static void
    displayTextureSizeInfo(void)
    {
        GLint maxTextureSize, maxTextureLevel;
        int maxWidth, maxHeight;

        glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxTextureSize);
        printf("GL_MAX_TEXTURE_SIZE: %d\n", maxTextureSize);

        maxTextureLevel = intFloorLog2(maxTextureSize);

        findMaxTexture(GL_RGB, 2, 1, 1, 0, 0, &maxWidth, &maxHeight);
        printf("RGB 1:1 (%d x %d)\n", maxWidth, maxHeight);

        findMaxTexture(GL_RGBA, 2, 2, 1, 0, 0, &maxWidth, &maxHeight);
        printf("RGBA 2:1 (%d x %d)\n", maxWidth, maxHeight);

        findMaxTexture(GL_RGB, 2, 1, 1, 0, maxTextureLevel, &maxWidth,
                                                             &maxHeight);
        printf("RGB 1:1 (%d x %d) mipmap\n", maxWidth, maxHeight);

        findMaxTexture(GL_LUMINANCE_ALPHA, 2, 1, 1, 1, 0, &maxWidth,
                                                           &maxHeight);
        printf("LUMINANCE_ALPHA 1:1 (%d x %d) w/border\n", maxWidth,
                                                            maxHeight);
    }
```

Extended Functions

See the reference pages of the following functions for related information:

glTexImage1D(), glTexImage2D(), glGetTexLevelParameteriv(), glTexEnvf(), glTexEnvi(), glTexEnvfv(), glTexEnviv(), glGetTexLevelParameterfv().

The Texture Object Extension

The texture object extension, EXT_texture_object, can improve the performance of programs that use multiple textures. It allows you to create as many textures as you need, and provides an efficient way to switch from one texture to another. This is similar to the texdef/texbind model of IRIS GL.

The extension also adds a new texture parameter, the priority, which you can use to tell the system which textures should have priority when it allocates hardware resources.

How Texture Objects Work

In OpenGL 1.0, only one texture can be associated with each of the texture targets GL_TEXTURE_1D and GL_TEXTURE_2D. If you want to render a scene that contains many 2D textures, you have to redefine the 2D texture using **glTexImage2D()** for each texture in the scene. If you want each texture to have different parameters, such as wrap modes and filters, you have to redefine those as well.

The texture object extension lets you create as many textures as you need. You associate a name (a positive number) with a texture object when you create it, and then you define the images and parameters of the texture. As you render your scene, bind the name of each desired texture object to the appropriate texture target. Since binding a texture takes less time than defining one, this is a more efficient way to switch from one texture to another.

Note: Texture objects and display lists are in different name spaces. However, if two contexts are sharing display lists, they also share texture objects.

Using Texture Objects

To create a texture object, bind an unused name to a texture target. The name is considered to be in use until you delete the texture. The following code fragment creates, defines, and binds texture objects with the names 1 and 2:

```
/* the first bind of 1 will create the texture object */
glBindTextureEXT( GL_TEXTURE_2D, 1 );
/* define the texture image and parameterss */
glTexImage2D( GL_TEXTURE_2D, 0, 4, 32, 32, 0, GL_RGBA, GL_BYTE, img1 );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );

/* the first bind of 2 will create the texture object */
glBindTextureEXT( GL_TEXTURE_2D, 2 );
/* define the texture image and parameters */
glTexImage2D( GL_TEXTURE_2D, 0, 4, 64, 64, 0, GL_RGBA, GL_BYTE, img2 );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );

/* bind each texture object as scene is rendered */
glBindTextureEXT( GL_TEXTURE_2D, 1 );
<draw some primitive>
glBindTextureEXT( GL_TEXTURE_2D, 2 );
<draw some primitive>
```

There is also an example program in the source tree that demonstrates how to use texture objects.

Texture Object Names

You can create a texture object with any name, as long as it isn't already the name of another texture object. To find out whether a particular name is in use, call **glIsTextureEXT()**. As a convenience, **glGenTexturesEXT()** returns a set of texture names that are known to be unused.

To delete one or more texture objects, call **glDeleteTexturesEXT()**. After a texture object is deleted, its name is freed; it has no contents, and its resources are returned to the system.

Editing and Querying Texture Objects

Once a texture object has been created, you can change its images and parameters at any time. You edit a texture object the same way you define it: you bind the texture object to a target and then call **glTexImage*()** or **glTexParameter*()**. These commands take a texture target as an argument, and they affect the texture object that is currently bound to that target. Similarly, you can query the parameters and images of a texture by calling **glGetTexParameter*()**, **glGetTexLevelParameter*()**, and **glGetTexImage()**, which also operate on the currently bound texture.

To find out which texture object is currently bound to each texture target, call **glGetIntegerv()** with GL_TEXTURE_1D_BINDING_EXT, GL_TEXTURE_2D_BINDING_EXT, or GL_TEXTURE_3D_BINDING_EXT (if your system supports 3D textures).

Texture Priorities and Residency

On some systems, textures must reside in hardware resources such as texture memory before they can be used. When you bind a texture, the system may need to swap out some other textures to make room, if the application's textures do not all fit in texture memory.

An application can guide the system in determining which textures should remain resident by specifying a priority for each texture using **glPrioritizeTexturesEXT()**. Each priority value is clamped to the range 0 to 1, with 0 indicating the lowest priority and hence the least likelihood of being resident, and 1 indicating the highest priority. You can also set the priority of a single texture by calling **glTexParameter*()** with GL_TEXTURE_PRIORITY_EXT. Note that the performance effects, if any, of setting texture priorities are entirely system dependent.

You can query whether a set of texture objects is resident by calling **glAreTexturesResidentEXT()**, and you can query whether a single texture object is resident by calling **glGetTexParameter*()** with GL_TEXTURE_RESIDENT_EXT.

Default Textures

You can still access the OpenGL 1.0 textures, now called the “default textures,” by binding 0 to a texture target. This is in fact the initial binding when you create a rendering context. When you create a texture object, its images and parameters are the same as the initial settings of the default textures.

You can set the priority of a default texture by calling **glTexParameter*()**; you cannot use **glPrioritizeTexturesEXT()** with a default texture. Similarly, you can query the residence of a default texture with **glGetTexParameter*()**, but you cannot do so with **glAreTexturesResidentEXT()**.

New and Extended Functions

New: glGenTexturesEXT(), glDeleteTexturesEXT(), glIsTextureEXT(), glBindTextureEXT() glPrioritizeTexturesEXT(), glAreTexturesResidentEXT().

Extended: glGetParameterTex*().

The Subtexture Extension

The subtexture extension, EXT_subtexture, lets you replace a subrectangle of an existing texture image without affecting the remaining portions of the image.

Why Use the Subtexture Extension?

Using a subtexture is the fastest way to update only the image of a texture. When you need to redefine only the image of the texture and keep all other parameters, using subtexture is much faster than using the **glTexImage*()** call.

The subtexture extension is especially useful under the following circumstances:

- When you're dealing with an image that doesn't fit into texture memory, you can use the extension to load only parts of the texture and replace them as appropriate.
- When you're using video frames as textures, none of the common video formats (NTSC, PAL, SECAM, HD-TV, and so on) have height and width that's a power of 2. This is problematic because height and width of those dimensions is required for textures. You therefore create a NULL texture whose dimensions are larger than the video frame and then load the video frame as a subtexture.
- When you're working with an animation, loading a large texture may decrease the frame rate. Instead, load several subtextures over multiple frames to maintain a high frame rate.

Using Subtextures

To load a subrectangle of a texture image, call **glTexSubImage1DEXT()**, **glTexSubImage2DEXT()**—prototype below—or **glTexSubImage3DEXT()**.

```
void glTexSubImage2DEXT(enum target, int level, int xoffset,int yoffset,
    sizei width, sizei height, enum format, enum type, const void* pixels)
```

- *target* specifies the existing texture image for which the subimage is being defined.
- *level*, *depth*, *format*, *type*, and *pixels* correspond to the arguments of **glTexImage2D()**; *width* and *height* are those of the subimage.
- *xoffset* and *yoffset* specify texel offsets in the x and y directions within the texture image being modified.

Using any uninitialized portion of a texture while drawing yields undefined results.

Using Null Images With Subtextures

It is sometimes useful to define the parameters of a texture image without actually initializing the contents of that image. For example, if you want to load a frame of an NTSC video as a texture, you need to first define a region larger than the video frame that meets the height and width requirements of textures (power of 2). Null images are also useful when you want to first define the parameters of a texture and later initialize the texture image using **glTexSubImage*()** calls.

To create a null image, call one of the texture creation functions, for example, **glTexImage2D()** with *pixels* set to the null pointer. The specified texture is created, but no pixels are processed. Using an uninitialized part of a texture yields undefined results.

The following code fragment creates a null image texture, then loads a subimage:

```
{
/* need to initialize "image" to something interesting */
    static unsigned char image[32][32][4];
/* create a 256 x 256 null texture */
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB5_EXT, 256, 256, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, NULL);
/* load a 32 x 32 subimage starting at 100,110 */
    glTexSubImage2DEXT(GL_TEXTURE_2D, 0, 100, 110, 32, 32,
                       GL_RGBA, GL_UNSIGNED_BYTE, image);
}
```

New Functions

glTexSubImage1DEXT(), glTexSubImage2DEXT(), glTexSubImage2DEXT().

The Copy Texture Extension

The copy texture extension, EXT_copy_texture, lets you load texture images directly from the framebuffer. You can replace either part of the texture image or the whole image. If your system supports the 3D texture extension, you can also load 2D slices of a 3D texture from the framebuffer. You can't load an entire 3D texture because the framebuffer is 2D.

Why Use the Copy Texture Extension?

The copy texture extension lets you do multi-pass rendering, using the full set of OpenGL rendering capabilities to generate a texture you can use for further drawing. For example, you can render a scene, copy the resulting image to texture memory, and then apply the texture to a triangle mesh. By changing the mesh, you can warp the scene, break it into pieces, wrap it around complex objects, and so on.

Copying textures is also the only means of directly using video frames as textures, in conjunction with the video source extension. See "The Video Source Extension" on page 183.

Copying Texture Images

The functions **glCopyTexImage1DEXT()** and **glCopyTexImage2DEXT()** copy image data from the color buffer specified by the current read buffer rather than accepting image data from host memory.

To copy an image from the framebuffer, call **glCopyTexImage2DEXT()**, **glCopyTexImage1DEXT()**, **glCopyTexSubImage1DEXT()**, or **glCopyTexSubImage2DEXT()**. The prototypes for the four functions are very similar; here's **glCopyTexImage2DEXT()** as an example:

```
void glCopyTexImage2DEXT( GLenum target,GLint level,GLenum internalformat,
         GLint x, GLint y, GLsizei width, GLsizei height, GLint border)
```

- *target* and *level* are equivalent to their **glTexImage2D()** counterparts, except that *target* does not accept GL_PROXY_TEXTURE_2D_EXT.
- *internalformat* is identical to the *components* parameter of **glTexImage2D()**, except that **glCopyTexImage2DEXT()** does not accept the values 1, 2, 3, and 4 but instead the internal formats defined by the texture extension, some of which are listed in Table 6-1 on page 87. For complete information, see the reference page.
- Each function reads image data from a color buffer region that is (*width+2*border_pixel)* wide and *(height+2*border_pixel*) high, and whose lower-left pixel has window coordinates x,y.

The pixel values are processed exactly as if **glCopyPixels()** had been called, but the process stops just before final conversion, that is, before clamping and conversion to fragments. At this point all pixel component values are clamped to [0,1], and then treated exactly as if the corresponding **glTexImage*()** or **glTexSubImage*()** had been called. Pixel ordering is such that lower x screen coordinates correspond to lower i (horizontal) texture coordinates, and lower y screen coordinates correspond to lower j (vertical) texture coordinates.

If the subtexture extension and the 3D texture extension are both supported, you can also use **glCopyTexSubImage3DEXT()**. See "The 3D Texture Extension" on page 100 for more information about 3D textures.

New Functions

glCopyTexImage1DEXT(), glCopyTexImage2DEXT(), glCopyTexSubImage1DEXT(), glCopyTexSubImage2DEXT(), glCopyTexSubImage3DEXT().

The 3D Texture Extension

The 3D texture extension, EXT_texture3D, defines 3-dimensional texture mapping and in-memory formats for 3D images, and adds pixel storage modes to support them.

3D textures can be thought of as an array of 2D textures, as illustrated in Figure 6-1.

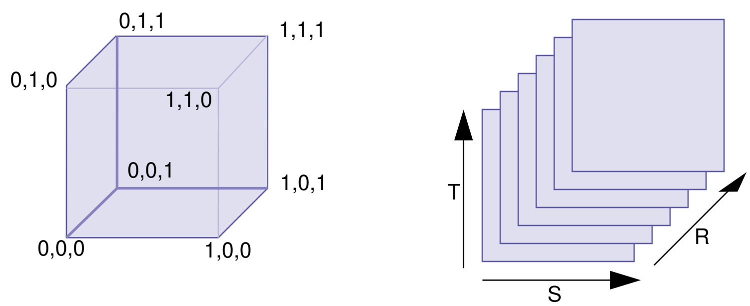


Figure 6-1 3D Texture

A 3D texture is mapped into (s,t,r) coordinates such that its lower left back corner is (0,0,0) and its upper right front corner is (1,1,1).

Why Use the 3D Texture Extension?

3D textures are useful for

- volume rendering and examining a 3D volume one slice at a time
- animating textured geometry, for example, people that move
- solid texturing, for example, wood, marble and so on
- eliminating distortion effects that occur when you try to map a 2D image onto 3D geometry

Texel values defined in a 3D coordinate system form a texture volume. You can extract textures from this volume by intersecting it with a 3D plane, as shown in Figure 6-2.

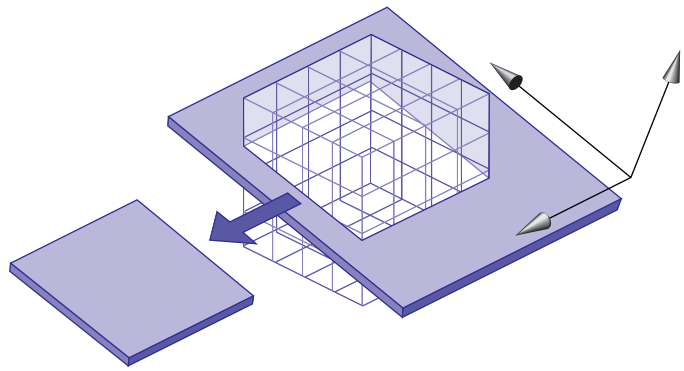


Figure 6-2 Extracting a Planar Texture From a 3D Texture Volume

The resulting texture, applied to a polygon, is the intersection of the volume and the plane. The orientation of the plane is determined from the texture coordinates of the vertices of the polygon.

Using 3D Textures

To create a 3D texture, use **glTexImage3DEXT()**. The function is defined like **glTexImage2DEXT()** but has a *depth* argument that specifies how many “slices” the texture consists of. Internal formats are supported.

The extension also provides the following additional features:

- **Pixel storage modes.** The extension extends the pixel storage modes by adding four new state variables:
  - GL_(UN)PACK_IMAGE_HEIGHT_EXT defines the height of the image the texture is read from, analogous to the GL_(UN)PACK_LENGTH variable for image width.
  - GL_(UN)PACK_SKIP_IMAGES_EXT determines an initial skip analogous to GL_(UN)PACK_SKIP_PIXELS and GL_(UN)PACK_SKIP_LINES.

  All four default to zero.
- **Texture wrap modes.** The functions **glTexParameter*()**, accept the additional token value GL_TEXTURE_WRAP_R_EXT.

It affects the R coordinate in the same way that GL_TEXTURE_WRAP_S affects the S coordinate and GL_TEXTURE_WRAP_T affects the T coordinate. The default value is GL_REPEAT.

- **Mipmapping.** Mipmapping for two-dimensional textures is discussed in the section "Multiple Levels of Detail," in Chapter 9, "Texture Mapping," of the *OpenGL Programming Guide*. Mipmapping for 3D textures works the same way: A 3D mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. The filtering options GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, and GL_LINEAR_MIPMAP_LINEAR apply to subvolumes instead of texels.
- **Proxy textures.** Use the proxy texture GL_PROXY_TEXTURE_3D_EXT to query an implementation's maximum configuration. For more information on proxy textures, see "The Texture Extension Proxy Mechanism" on page 89.
- **Querying.** Use the following call to query the 3D texture:

  ```
  glGetTexImage(GL_TEXTURE_3D_EXT)
  ```
- Subvolumes of the 3D texture can be replaced using **glTexSubImage3DEXT()** and **glCopyTexSubImage3DEXT()** (see "The Subtexture Extension" on page 96).

3D Texture Example Program

The code fragment presented in this section illustrates the use of the extension. The complete program is included in the example source tree.

```
/*
 * tex3d - simple 3D texturing program
 *
 * Shows a 3D texture by drawing slices through it.
 */
/* compile: cc -o tex3d tex3d.c -lGL -lX11 */

#include <GL/glx.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>

static int attributeList[] = { GLX_RGBA, None };
```

```
unsigned int tex[64][64][64];

/* generate a simple 3D texture */
static void
make_texture(void) {
    int i, j, k;
    unsigned int *p = &tex[0][0][0];

    for (i=0; i<64; i++) {
        for (j=0; j<64; j++) {
            for (k=0; k<64; k++) {
                if (i < 10 || i > 48 ||
                    j < 10 || j > 48 ||
                    k < 10 || k > 48) {
                    if (i < 2 || i > 62 ||
                        j < 2 || j > 62 ||
                        k < 2 || k > 62) {
                        *p++ = 0x00000000;
                    } else {
                        *p++ = 0xff80ffff;
                    }
                } else {
                    *p++ = 0x000000ff;
                }
            }
        }
    }
}

static void
init(void) {
    make_texture();
    glEnable(GL_TEXTURE_3D_EXT);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glClearColor(0.2,0.2,0.5,1.0);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

    glMatrixMode(GL_PROJECTION);
    gluPerspective(60.0, 1.0, 1.0, 100.0 );
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.,0.,-3.0);
    glMatrixMode(GL_TEXTURE);
```

```
    /* Similar to defining a 2D texture, but note the setting of the */
    /* wrap parameter for the R coordinate.  Also, for 3D textures   */
    /* you probably won't need mipmaps, hence the linear min filter. */
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_MIN_FILTER,
                                                  GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R_EXT,
                                                   GL_CLAMP);
    glTexImage3DEXT(GL_TEXTURE_3D_EXT, 0, 4, 64, 64, 64, 0,
                    GL_RGBA, GL_UNSIGNED_BYTE, tex);
}

#define NUMSLICES 256

static void
draw_scene(void) {
    int i;
    float r, dr, z, dz;

    glColor4f(1, 1, 1, 1.4/NUMSLICES);
    glClear(GL_COLOR_BUFFER_BIT);

    /* Display the entire 3D texture by drawing a series of quads */
    /* that  slice through the texture coordinate space.  Note that */
    /* the transformations below are applied to the texture matrix, */
    /* not the modelview matrix. */

    glLoadIdentity();
    /* center the texture coords around the [0,1] cube */
    glTranslatef(.5,.5,.5);
    /* a rotation just to make the picture more interesting */
    glRotatef(45.,1.,1.,.5);

    /* to make sure that the texture coords, after arbitrary */
    /* rotations, still fully contain the [0,1] cube, make them span */
    /* a range sqrt(3)=1.74 wide */
    r = -0.87; dr = 1.74/NUMSLICES;
    z = -1.00; dz = 2.00/NUMSLICES;
    for (i=0; i < NUMSLICES; i++) {
        glBegin(GL_TRIANGLE_STRIP);
        glTexCoord3f(-.87,-.87,r); glVertex3f(-1,-1,z);
        glTexCoord3f(-.87, .87,r); glVertex3f(-1, 1,z);
        glTexCoord3f( .87,-.87,r); glVertex3f( 1,-1,z);
```

```
        glTexCoord3f( .87, .87,r); glVertex3f( 1, 1,z);
        glEnd();
        r += dr;
        z += dz;
    }
}

/* process input and error functions and main(), which handles window
 * setup, go here.
 */
```

New Functions

glTexImage3DEXT().

The Texture Color Table Extension

The texture color table extension, SGI_texture_color_table, adds a color lookup table to the texture mechanism. The table is applied to the filtered result of a texture lookup before that result is used in the texture environment equations.

Why Use a Texture Color Table?

Here are two example situations in which the texture color table extension is useful:

- **Volume rendering.** You can store something other than color in the texture (for example, a physical attribute like bone density) and use the table to map that density to an RGB color. This is useful if you want to display just that physical attribute (for example, bone density) and also if you want to distinguish between that attribute and another (for example, muscle density). You can selectively replace the table to display different features. Note that updating the table can be faster than updating the texture. (This technique also called “false color imaging” or “segmentation”).
- **Representing shades (gamut compression).** If you need to display a high color-resolution image using a texture with low color-component resolution, the result is often unsatisfactory. A 16-bit texel doesn’t offer a lot of shades for each color, because each color component has to be evenly spaced between black and the strongest shade of the color. If an image contains several shades of light blue but no

dark blue, for example, the on-screen image can't represent that easily because only a limited number of shades of blue, many of them dark, are available. When using a color table, you can "stretch" the colors.

Using Texture Color Tables

To use a texture color table, define a color table, as described in "The Color Table Extension" on page 136. Use GL_TEXTURE_COLOR_TABLE_SGI as the value for the *target* parameter of the various commands, keeping in mind the following points:

- The table size, specified by the *width* parameter of **glColorTableSGI(),** is limited to powers of two.
- The maximum table size is 256 entries.
- Use GL_PROXY_TEXTURE_COLOR_TABLE_SGI to find out whether there is enough room for the texture color table in exactly the manner described in "The Texture Extension Proxy Mechanism" on page 89.

The following code fragment loads a table that inverts a texture. It uses a GL_LUMINANCE external format table to make identical R, G, and B mappings.

```
loadinversetable()
{
    static unsigned char table[256];
    int i;

    for (i = 0; i < 256; i++) {
        table[i] = 255-i;
    }

    glColorTableSGI(GL_TEXTURE_COLOR_TABLE_SGI, GL_RGBA8_EXT,
                    256, GL_LUMINANCE, GL_UNSIGNED_BYTE, table);
    glEnable(GL_TEXTURE_COLOR_TABLE_SGI);
}
```

The Sharpen Texture Extension

This section discusses the sharpen texture extension, SGIS_sharpen_texture. This extension and the detail texture extension (see "The Detail Texture Extension" on page 112) are useful in situations where you want to maintain good image quality when a texture must be magnified for close-up views.

When a textured surface is viewed close up, the magnification of the texture can cause blurring. One way to reduce blurring is to use a higher-resolution texture for the close-up view, at the cost of extra storage. The sharpen texture extension offers a way to keep the image crisp without increasing texture storage requirements.

Sharpen texture works best when the high-frequency information is used to represent edge information, for example:

- In a stop sign, the edges of the letters have distinct outlines, and bilinear magnification normally causes the letters to blur. Sharpen texture keeps the edges crisp.
- In a tree texture, the alpha values are high inside the outline of the tree and low outside the outline (where the background shows through). Bilinear magnification normally causes the outline of the tree to blur. Sharpen texture, applied to the alpha component, keeps the outline crisp.

Sharpen texture works by extrapolating from mipmap levels 1 and 0 to create a magnified image that has sharper features than either mipmap.

Using the Sharpen Texture Extension

This section first explains how to use the sharpen texture extension to sharpen the component of your choice. It then gives some background information about how the extension works and explains how you can customize the LOD extrapolation function.

How to Use the Sharpen Texture Extension

You can use the extension to sharpen the alpha component, the color components, or both, depending on the magnification filter. To specify sharpening, use one of the magnification filters in Table 6-3.

Table 6-3 Magnification Filters for Sharpen Texture

GL_TEXTURE_MAG_FILTER	Alpha	Red, Green, Blue
GL_LINEAR_SHARPEN_SGIS	sharpen	sharpen
GL_LINEAR_SHARPEN_COLOR_SGIS	bilinear	sharpen
GL_LINEAR_SHARPEN_ALPHA_SGIS	sharpen	bilinear

For example, suppose that a texture contains a picture of a tree in the color components, and the opacity in the alpha component. To sharpen the outline of the tree, use

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                               GL_LINEAR_SHARPEN_ALPHA_SGIS);
```

How Sharpen Texture Works

When OpenGL applies a texture to a pixel, it computes a level of detail (LOD) factor that represents the amount by which the base texture (that is, level 0) must be scaled. LOD *n* represents a scaling of 2^{-n}. For example, if OpenGL needs to magnify the base texture by a factor of 4 in both S and T, the LOD is -2. Note that magnification corresponds to negative values of LOD.

To produce a sharpened texel at level-of-detail *n*, OpenGL adds the weighted difference between the texel at LOD 0 and LOD 1 to LOD 0; that is:

```
LODn = LOD0 + weight(n) * (LOD0 - LOD1)
```

The variables are defined as follows:

n	level-of-detail
weight(*n*)	LOD extrapolation function
LOD0	base texture
LOD1	texture at mipmap level 1

By default, OpenGL uses a linear extrapolation function, where weight(n) = $-n/4$. You can customize the LOD extrapolation function by specifying its control points, as discussed in the next section.

Customizing the LOD Extrapolation Function

With the default linear LOD extrapolation function, the weight may be too large at high levels of magnification, that is, as n becomes more negative. This can result in so much extrapolation that noticeable bands appear around edge features, an artifact known as "ringing." In this case, it's useful to create a nonlinear LOD extrapolation function.

Figure 6-3 shows LOD extrapolation curves as a function of magnification factors. The curve on the left is the default linear extrapolation, where weight(n) = $-n/4$. The curve on the right is a nonlinear extrapolation, where the LOD extrapolation function is modified to control the amount of sharpening so that less sharpening is applied as the magnification factor increases. The function is defined for n less than or equal to 0.

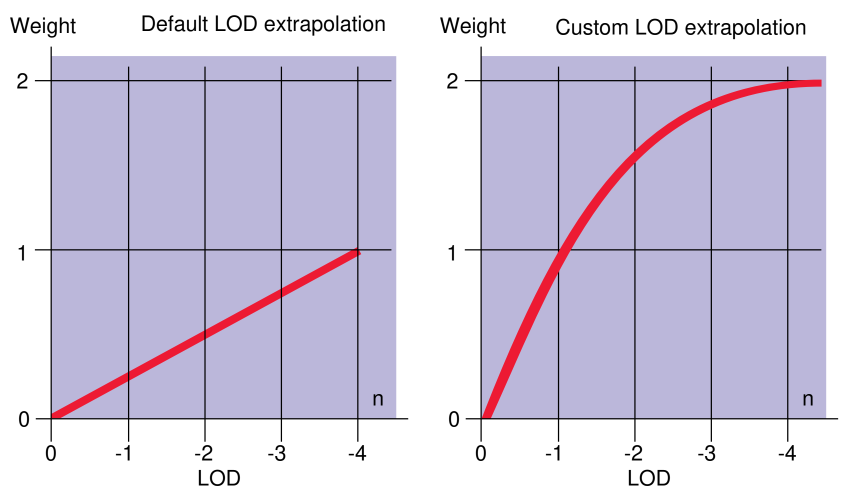


Figure 6-3 LOD Extrapolation Curves

Use **glSharpenTexFuncSGIS()** to specify control points for shaping the LOD extrapolation function. Each control point contains a pair of values; the first value specifies the LOD, and the second value specifies a weight multiplier for that magnification level. (Remember that the LOD values are negative.)

For example, to gradually ease the sharpening effect, use a nonlinear LOD extrapolation curve—as shown on the right in Figure 6-3—with these control points:

```
GLfloat points[] = {
     0., 0.,
-    1., 1.,
-    2., 1.7,
-    4., 2.
};
glSharpenTexFuncSGIS(GL_TEXTURE_2D, 4, points);
```

Note that how these control points determine the function is system dependent. For example, your system may choose to create a piecewise linear function, a piecewise quadratic function, or a cubic function. However, regardless of which kind of function is chosen, the function will pass through the control points.

Using Sharpen Texture and Texture Object

If you are using the texture object extension, each texture object contains its own LOD extrapolation function and magnification filter. Setting the function or the filter therefore affects only the texture object that is currently bound to the texture target.

Sharpen Texture Example Program

Example 6-2 illustrates the use of the sharpen texture example program. Because of space limitations, the sections dealing with X Window System setup and some of the keyboard input are omitted. The complete example is included in the source tree as *sharpen.c*.

Example 6-2 Sharpen Texture Example

```
/* tree texture: high alpha in foreground, zero alpha in background */
#define B 0x00000000
#define F 0xA0A0A0ff
unsigned int tex[] = {
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,F,F,B,B,B,B,B,B,B,
    B,B,B,B,B,B,B,F,F,B,B,B,B,B,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,F,F,F,F,F,F,B,B,B,B,B,
    B,B,B,B,B,F,F,F,F,F,F,B,B,B,B,B,
    B,B,B,B,F,F,F,F,F,F,F,F,B,B,B,B,
    B,B,B,B,F,F,F,F,F,F,F,F,B,B,B,B,
    B,B,B,F,F,F,F,F,F,F,F,F,F,B,B,B,
    B,B,B,F,F,F,F,F,F,F,F,F,F,B,B,B,
    B,B,F,F,F,F,F,F,F,F,F,F,F,F,B,B,
    B,B,F,F,F,F,F,F,F,F,F,F,F,F,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,B,F,F,F,F,B,B,B,B,B,B,
    B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,B,
};

static void
init(void) {
    glEnable(GL_TEXTURE_2D);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(60.0, 1.0, 1.0, 10.0 );
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.,0.,-2.5);

    glColor4f(0,0,0,1);
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    /* sharpening just alpha keeps the tree outline crisp */
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR_SHARPEN_ALPHA_SGIS);
    /* generate mipmaps; levels 0 and 1 are needed for sharpening */
    gluBuild2DMipmaps(GL_TEXTURE_2D, 4, 16, 16, GL_RGBA,
                      GL_UNSIGNED_BYTE, tex);
}
```

```
static void
draw_scene(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLE_STRIP);
        glTexCoord2f( 0, 1); glVertex2f(-1,-1);
        glTexCoord2f( 0, 0); glVertex2f(-1, 1);
        glTexCoord2f( 1, 1); glVertex2f( 1,-1);
        glTexCoord2f( 1, 0); glVertex2f( 1, 1);
    glEnd();
    glFlush();
}
```

New Functions

glSharpenTexFuncSGIS(), glGetSharpenTexFuncSGIS().

The Detail Texture Extension

This section discusses the detail texture extension, SGIS_detail_texture, which like the sharpen texture extension (see "The Sharpen Texture Extension" on page 107) is useful in situations where you want to maintain good image quality when a texture must be magnified for close-up views.

Ideally, programs should always use textures that have high enough resolution to allow magnification without blurring. High-resolution textures maintain realistic image quality for both close-up and distant views. For example, in a high-resolution road texture, the large features—such as potholes, oil stains, and lane markers that are visible from a distance—as well as the asphalt of the road surface, look realistic no matter where the viewpoint is.

Unfortunately, a high-resolution road texture with that much detail may be as large as 2K x 2K, which may exceed the texture storage capacity of the system. Making the image close to or equal to the maximum allowable size still leaves little or no memory for the other textures in the scene.

The detail texture extension provides a solution for representing a 2K x 2K road texture with smaller textures. Detail texture works best for a texture with high-frequency information that is not strongly correlated to its low-frequency information. This occurs in images that have a uniform color and texture variation throughout, such as a field of grass or a wood panel with a uniform grain. If high-frequency information in your texture is used to represent edge information (for example, a stop sign or the outline of a tree) consider the sharpen texture extension (see "The Sharpen Texture Extension" on page 107).

Using the Detail Texture Extension

Because the high-frequency detail in a texture (for example, a road) is often virtually the same across the entire texture, the detail from an arbitrary portion of the texture image can be used as the detail across the entire image.

When you use the detail texture extension, the high-resolution texture image is represented by the combination of a low-resolution texture image and a small high-frequency detail texture image (the detail texture). OpenGL combines these two images during rasterization to create an approximation of the high-resolution image.

This section first explains how to create the detail texture and the low resolution texture that are used by the extension, then briefly looks at how detail texture works and how to customize the LOD interpolation function, which controls how OpenGL combines the two textures.

Creating a Detail Texture and a Low-Resolution Texture

This section explains how to convert a high-resolution texture image into a detail texture and a low-resolution texture image. For example, for a 2K x 2K road texture, you may want to use a 512 x 512 low-resolution base texture and a 256 x 256 detail texture. Follow these steps to create the textures:

1. Make the low-resolution image using *izoom* or another resampling program to make the low-resolution image by shrinking the high-resolution image by 2^n.

   In this example, n is 2, so the resolution of the low-resolution image is 512 x 512. This band-limited image has the two highest-frequency bands of the original image removed from it.

2. Create the subimage for the detail texture using *subimage* or another tool to select a 256 x 256 region of the original high-resolution image, whose n highest-frequency bands are characteristic of the image as a whole. (For example, rather than choosing a subimage from the lane markings or a road, choose an area in the middle of a lane.)

3. Optionally, make this image self-repeating along its edges to eliminate seams.

4. Create a blurry version of the 256 × 256 subimage as follows:
   - First shrink the 256 × 256 subimage by 2^n, to 64 × 64.
   - Then scale the resulting image back up to 256 × 256.

   The image is blurry because it is missing the two highest-frequency bands present in the two highest levels of detail.
5. Subtract the blurry subimage from the original subimage. This difference image—the detail texture—has only the two highest frequency bands.
6. Define the low-resolution texture (the base texture created in Step 1) with the GL_TEXTURE_2D target and the detail texture (created in Step 5) with the GL_DETAIL_TEXTURE_2D_SGIS target. In the road example, you would use

   ```
   GLvoid *detailtex, *basetex;
   glTexImage2D(GL_DETAIL_TEXTURE_2D_SGIS, 0, 4, 256, 256, 0, GL_RGBA,
               GL_UNSIGNED_BYTE, detailtex);
   glTexImage2D(GL_TEXTURE_2D, 0, 4, 512, 512, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, basetex);
   ```

   The internal format of the detail texture and the base texture must match exactly.
7. Set the GL_DETAIL_TEXTURE_LEVEL_SGIS parameter to specify the level at which the detail texture resides. In the road example, the detail texture is level -2 (since the original 2048 x 2048 texture is two levels below the 512 x 512 base texture):

   ```
   glTexParameteri(GL_TEXTURE_2D, GL_DETAIL_TEXTURE_LEVEL_SGIS, -2);
   ```

   Since the actual detail texture supplied to OpenGL is 256 x 256, OpenGL replicates the detail texture as necessary to fill a 2048 x 2048 texture. In this case, the detail texture repeats eight times in S and in T.

   Note that the detail texture level is set on the GL_TEXTURE_2D target, not on GL_DETAIL_TEXTURE_2D_SGIS.
8. Set the magnification filter to specify whether the detail texture is applied to the alpha or color component, or both. Use one of the filters in Table 6-3. For example, to apply the detail texture to both alpha and color components, use

   ```
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR_DETAIL_SGIS);
   ```

Note that the magnification filter is set on the GL_TEXTURE_2D target, not on GL_DETAIL_TEXTURE_2D_SGIS.

Table 6-4 Magnification Filters for Detail Texture

GL_TEXTURE_MAG_FILTER	Alpha	Red, Green, Blue
GL_LINEAR_DETAIL_SGIS	detail	detail
GL_LINEAR_DETAIL_COLOR_SGIS	bilinear	detail
GL_LINEAR_DETAIL_ALPHA_SGIS	detail	bilinear

How Detail Texture Is Computed

For each pixel that OpenGL textures, it computes an LOD-based factor that represents the amount by which the base texture (that is, level 0) is scaled. LOD *n* represents a scaling of 2^{-n}. Negative values of LOD correspond to magnification of the base texture.

To produce a detailed textured pixel at level-of-detail *n*, OpenGL uses one of the two formulas shown in Table 6-5, depending on the detail texture mode.

Table 6-5 How Detail Texture Is Computed

GL_DETAIL_TEXTURE_MODE_SGIS	Formula
GL_ADD	LODn = LOD0 + weight(*n*) ∗ DET
GL_MODULATE	LODn = LOD0 + weight(*n*) * DET * LOD0

The variables in the formulas are defined as follows:

n	level of detail
weight(*n*)	detail function
LOD0	base texture
DET	detail texture

For example, to specify GL_ADD as the detail mode, use

```
glTexParameteri(GL_TEXTURE_2D, GL_DETAIL_TEXTURE_MODE_SGIS, GL_ADD);
```

Customizing the Detail Function

In the road example, the 512 x 512 base texture is LOD 0. The detail texture combined with the base texture represents LOD -2, which is called the maximum-detail texture.

By default, OpenGL performs linear interpolation between LOD 0 and LOD -2 when a pixel's LOD is between 0 and -2. Linear interpolation between more than one LOD can result in aliasing. To minimize aliasing between the known LODs, OpenGL lets you specify a nonlinear LOD interpolation function.

Figure 6-4 shows the default linear interpolation curve and a nonlinear interpolation curve that minimizes aliasing when interpolating between two LODs.

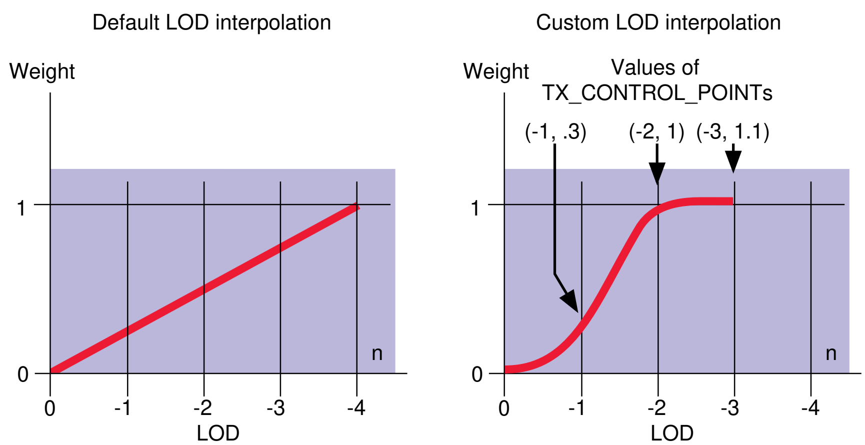


Figure 6-4 LOD Interpolation Curves

The basic strategy is to use very little of the detail texture until the LOD is within one LOD of the maximum-detail texture. More of the information from the detail texture can be used as the LOD approaches LOD -2. At LOD -2, the full amount of detail is used, and the resultant texture exactly matches the high-resolution texture.

Use **glDetailTexFuncSGIS()** to specify control points for shaping the LOD interpolation function. Each control point contains a pair of values; the first value specifies the LOD, and the second value specifies the weight for that magnification level.

The following control points can be used to create a nonlinear interpolation function (as shown above in Figure 6-4):

```
GLfloat points[] = {
     0.0, 0.0,
    -1.0, 0.3,
    -2.0, 1.0,
    -3.0, 1.1
};
glDetailTexFuncSGIS(GL_TEXTURE_2D, 4, points);
```

Note that how these control points determine a function is system dependent. For example, your system may choose to create a piecewise linear function, a piecewise quadratic function, or a cubic function. However, regardless of which kind of function is chosen, the function passes through the control points.

Using Detail Texture and Texture Object

If you are using the texture object extension, the base texture and the detail texture are separate texture objects. You can bind any base texture object to GL_TEXTURE_2D and any detail texture object to GL_DETAIL_TEXTURE_2D_SGIS. Each base texture object contains its own detail mode, magnification filter, and LOD interpolation function. Setting these parameters therefore affects only the texture object that is currently bound to GL_TEXTURE_2D. (If you set these parameters on the detail texture object, they are ignored.)

Detail Texture Example Program

Example 6-3 is a code fragment taken from a simple detail texture example program. The complete example is included in the source tree as *detail.c.*

Example 6-3 Detail Texture Example

```
unsigned int tex[128][128];
unsigned int detailtex[256][256];

static void
make_textures(void) {
    int i, j;
    unsigned int *p;

    /* base texture is solid gray */
```

```
    p = &tex[0][0];
    for (i=0; i<128*128; i++) *p++ = 0x808080ff;

    /* detail texture is a yellow grid over a gray background     */
    /* this artificial detail texture is just a simple example    */
    /* you should derive a real detail texture from the original  */
    /* image as explained in the text.                            */
    p = &detailtex[0][0];
    for (i=0; i<256; i++) {
        for (j=0; j<256; j++) {
            if (i%8 == 0 || j%8 == 0) {
                *p++ = 0xffff00ff;
            } else {
                *p++ = 0x808080ff;
            }
        }
    }
}

static void
init(void) {
    make_textures();

    glEnable(GL_TEXTURE_2D);
    glMatrixMode(GL_PROJECTION);
    gluPerspective(90.0, 1.0, 0.3, 10.0 );
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0.,0.,-1.5);

    glClearColor(0.0, 0.0, 0.0, 1.0);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    /* NOTE: parameters are applied to base texture, not the detail */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR_DETAIL_SGIS);
    glTexParameteri(GL_TEXTURE_2D, GL_DETAIL_TEXTURE_LEVEL_SGIS, -1);
    glTexImage2D(GL_TEXTURE_2D,
                 0, 4, 128, 128, 0, GL_RGBA, GL_UNSIGNED_BYTE, tex);
    glTexImage2D(GL_DETAIL_TEXTURE_2D_SGIS,
                 0, 4, 256, 256, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                 detailtex);
}
```

```
static void
draw_scene(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLE_STRIP);
        glTexCoord2f( 0, 0); glVertex3f(-1,-0.4, 1);
        glTexCoord2f( 0, 1); glVertex3f(-1,-0.4,-1);
        glTexCoord2f( 1, 0); glVertex3f( 1,-0.4, 1);
        glTexCoord2f( 1, 1); glVertex3f( 1,-0.4,-1);
    glEnd();
    glFlush();
}
```

New Functions

glDetailTexFuncSGIS(), glGetDetailTexFuncSGIS().

Texture Clamp Extensions

This section first provides some background information on texture clamping. It then looks at reasons for using the texture clamping extensions and explains how to use them. The two extensions are

- the texture edge clamp extension, SGIS_texture_edge_clamp
- the texture border clamp extension, SGIS_texture_border_clamp

Texture clamping is especially useful for nonrepeating textures.

Texture Clamping Background Information

OpenGL 1.0 provides clamping of texture coordinates: Any values greater than 1.0 are set to 1.0, any values less than 0.0 are set to 0.0. Clamping is useful for applications where you want a single copy of the texture to be mapped onto a large surface. Clamping is discussed in detail in the section "Repeating and Clamping Textures" in Chapter 9, "Texture Mapping," of the *OpenGL Programming Guide.*

Why Use the Texture Clamp Extensions?

When a texture coordinate is clamped using the default OpenGL 1.0 algorithm, and a GL_LINEAR filter or one of the LINEAR mipmap filters is used, the texture sampling filter straddles the edge of the texture image, taking half its sample values from within the texture image and the other half from the texture border.

It is sometimes desirable to alter the default behavior of OpenGL texture clamping operations as follows:

- Clamp a texture without requiring a border or a constant border color. This is possible with the texture clamping algorithm provided by the texture edge-clamp extension. GL_CLAMP_TO_EDGE_SGIS clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel.

  When used with a GL_NEAREST or a GL_LINEAR filter, the color returned when clamping is derived only from texels at the edge of the texture image.
- Clamp a texture to the border color, rather than to an average of the border and edge colors. This is possible with the texture border-clamp extension. GL_CLAMP_TO_BORDER_SGIS clamps texture coordinates at all mipmap levels:

  GL_NEAREST and GL_LINEAR filters return the color of the border texels.

  This mode is well-suited for using projective textures such as spotlights.

Both clamping extensions are supported for one-, two-, and three-dimensional textures.

Using the Texture Clamp Extensions

To specify texture clamping, call **glTexParameteri()**:

- Set *target* to GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D_EXT.
- Set *pname* to GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, or GL_TEXTURE_WRAP_R_EXT.
- Set *param* to
  – GL_CLAMP_TO_EDGE_SGIS for edge clamping
  – GL_CLAMP_TO_BORDER_SGIS for border clamping

The Texture LOD Extension

The texture LOD extension, SGIS_texture_lod, imposes constraints on the texture LOD parameter. Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. By providing separate, continuous clamping of the LOD parameter, the extension makes it possible to avoid "popping" artifacts when higher-resolution images are provided.

To achieve this, the extension imposes the following constraints:

- It clamps LOD to a specific floating point range.
- It limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

To understand the issues discussed in this section, you should be familiar with the issues discussed in the sections "Multiple Levels of Detail" and "Controlling Filtering" in Chapter 9, "Texture Mapping," of the *OpenGL Programming Guide.*

Specifying a Minimum or Maximum Level of Detail

To specify a minimum or maximum level of detail for a specific texture, call **glTexParameter*()** and set

- *target* to GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D_EXT
- *pname* to GL_TEXTURE_MIN_LOD_SGIS or GL_TEXTURE_MAX_LOD_SGIS
- *param* to (or *params* pointing to) the new value

LOD is clamped to the specified range before its use in the texturing process. Whether the minification or magnification filter is used depends on the clamped LOD.

Specifying Image Array Availability

The *OpenGL Specification* describes a "complete" set of mipmap image arrays at levels 0 (zero) through p, where p is a well-defined function of the dimensions of the level 0 image.

This extension lets you redefine any image level as the base level (or maximum level). This is useful, for example, if your application runs under certain time constraints, and you want to make it possible for the application to load as many levels of detail as possible but stop loading and continue processing, choosing from the available ones after a certain period of time has elapsed. Availability in that case does not depend on what's explicitly specified in the program but on what could be loaded in a specified time.

To set a new base (or maximum) level, call **glTexParameteri()**, **glTexParemeterf()**, **glTexParameteriv()**, or **glTexParameterfv()** and set

- *target* to GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D_EXT
- *pname* to
  - GL_TEXTURE_BASE_LEVEL_SGIS to specify a base level
  - GL_TEXTURE_MAX_LEVEL_SGIS to specify a maximum level
- *param* to (or *params* pointing to) the desired value

Chapter 7

Imaging and Blending Extensions

This chapter discusses imaging and blending extensions. After some introductory information, it looks at each extension in some detail. You learn about:

- "Introduction to Imaging and Blending Extensions" on page 123
- "The ABGR Extension" on page 125
- "The Packed Pixels Extension" on page 125
- "The Color Matrix Extension" on page 128
- "The Convolution Extension" on page 129
- "The Histogram and Minmax Extensions" on page 132
- "The Interlace Extension" on page 139
- "The Color Table Extension" on page 136
- "The Interlace Extension" on page 139
- "Blending Extensions" on page 140

Introduction to Imaging and Blending Extensions

This section discusses where extensions are in the imaging pipeline established by OpenGL 1.0; it also lists the commands to which imaging extensions apply.

Where Extensions Are in the Imaging Pipeline

The OpenGL 1.0 imaging pipeline is shown in the *OpenGL Programming Guide* in the illustration "Drawing Pixels with glDrawPixels*()" in Chapter 8, "Drawing Pixels, Bitmaps, Fonts, and Images."

You can see how, after the scale and bias operations and after the shift and offset operations, color conversion (LUT in Figure 7-1 below) takes place with a lookup table.

After that, the extension modules may be applied. Unless the histogram or minmax extensions were called to collect information only, pixel processing continues, as shown in the *OpenGL Programming Guide.*

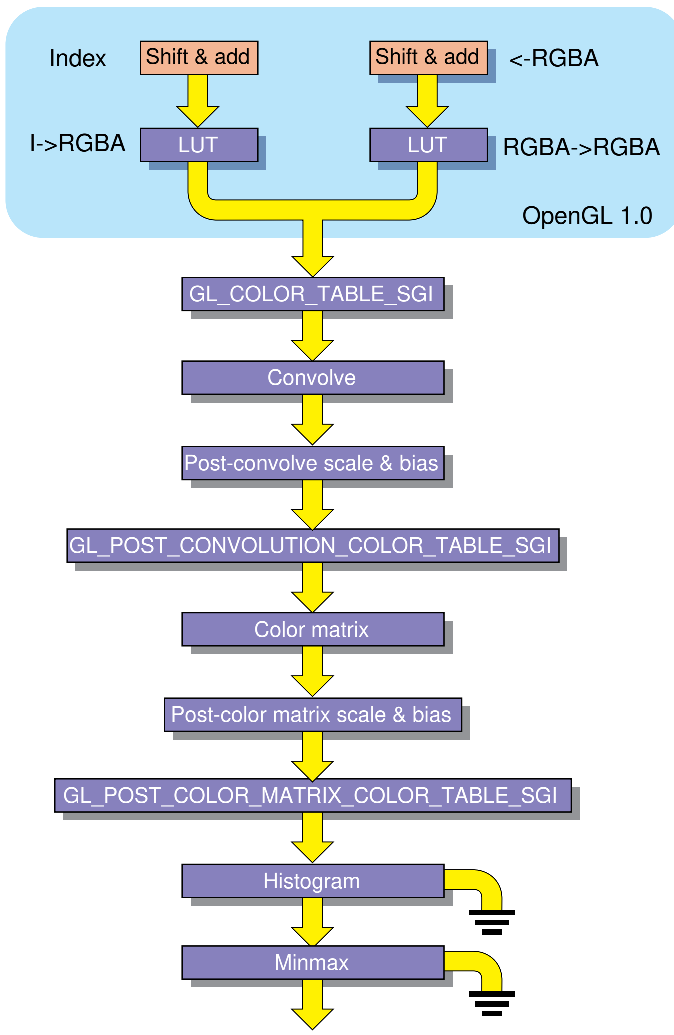


Figure 7-1 How Extensions Fit in the Imaging Pipeline

Functions Affected by Imaging Extensions

Imaging extensions affect all functions that are affected by the pixel transfer modes (see "Storing, Transforming, and Mapping Pixels" in Chapter 8, "Drawing Pixels, Bitmaps, Fonts, and Images," of the *OpenGL Programming Guide).* In general, these are all the commands that draw and copy pixels or define texture images and all commands that read pixels or textures back to host memory.

The ABGR Extension

The ABGR extension, EXT_abgr, extends the list of host-memory color formats by an alternative to the RGBA format that uses reverse component order. The ABGR component order matches the cpack IRIS GL format on big-endian machines. This is the most convenient way to use an ABGR source image with OpenGL.

Note that the ABGR extension provides the best performance on the following graphics systems: Starter, XZ, Elan, XS24, Extreme.

To use this extension, call **glDrawPixels()**, **glGetTexImage()**, **glReadPixels()**, and **glTexImage*()** with GL_ABGR_EXT as the value of the *format* parameter.

The following code fragment illustrates the use of the extension:

```
/*
 *  draw a 32x32 pixel image at location 10, 10 using an ABGR source
 *  image. "image" *should* point to a 32x32 ABGR UNSIGNED BYTE image
 */

{
    unsigned char *image;

    glRasterPos2f(10, 10);
    glDrawPixels(32, 32, GL_ABGR_EXT, GL_UNSIGNED_BYTE, image);
}
```

The Packed Pixels Extension

The packed pixels extension, EXT_packed_pixels, provides support for packed pixels in host memory. A packed pixel is represented entirely by one unsigned byte, unsigned short, or unsigned integer. The fields within the packed pixel are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes, such as GL_PACK_SKIP_PIXELS, GL_PACK_ROW_LENGTH, and so on, and their unpacking counterparts, all work correctly with packed pixels.

Why Use the Packed Pixels Extension?

The packed pixels extension lets you store images more efficiently by providing additional pixel types you can use when reading and drawing pixels or loading textures. Packed pixels may also be faster than unpacked pixels in some cases because less bandwidth (or less processing) is required to transfer them to and from the graphics hardware.

In addition, some of the types defined by this extension match the internal texture formats (see “The Texture Extension” on page 85) so less processing is required to transfer texture images to texture memory.

Using Packed Pixels

To use packed pixels, provide one of the types listed in Table 7-1 as the *type* parameter to **glDrawPixels()**, **glReadPixels()**, and so on.

Table 7-1 Types That Use Packed Pixels

Parameter Token Value	GL Data Type
GL_UNSIGNED_BYTE_3_3_2_EXT	GLubyte
GL_UNSIGNED_SHORT_4_4_4_4_EXT	GLushort
GL_UNSIGNED_SHORT_5_5_5_1_EXT	GLushort
GL_UNSIGNED_INT_8_8_8_8_EXT	GLuint
GL_UNSIGNED_INT_10_10_10_2_EXT	GLuint

The already available types for **glReadPixels()**, **glDrawPixels()**, and so on are listed in Table 8-2 “Data Types for glReadPixels or glDrawPixels,” in the *OpenGL Programming Guide*.

What the Different Pixel Types Mean

Each packed pixel type includes a base type, for example GL_UNSIGNED_BYTE, and a field width (for example, 3_3_2):

- The base type, GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT, determines the type of "container" into which each pixel's color components are packed.
- The field widths, 3_3_2, 4_4_4_4, 5_5_5_1, 8_8_8_8, or 10_10_10_2, determine the sizes (in bits) of the fields that contain a pixel's color components. The field widths are matched to the components in the pixel format, in left-to-right order.

  For example, if a pixel has the type GL_UNSIGNED_BYTE_3_3_2_EXT and the format GL_RGB, the pixel is contained in an unsigned byte, the red component occupies three bits, the green component occupies three bits, and the blue component occupies two bits.

  The fields are packed tightly into their container, with the leftmost field occupying the most-significant bits and the rightmost field occupying the least-significant bits.

Because of this ordering scheme, integer constants (particularly hexadecimal constants) can be used to specify pixel values in a readable and system-independent way. For example, a packed pixel with type GL_UNSIGNED_SHORT_4_4_4_4_EXT, format GL_RGBA, and color components red == 1, green == 2, blue == 3, alpha == 4 has the value 0x1234.

The ordering scheme also allows packed pixel values to be computed with system-independent code. For example, if there are four variables (red, green, blue, alpha) containing the pixel's color component values, a packed pixel of type GL_UNSIGNED_INT_10_10_10_2_EXT and format GL_RGBA can be computed with the following C code:

```
GLuint pixel, red, green, blue, alpha;
pixel = (red << 22) | (green << 12) | (blue << 2) | alpha;
```

While the source code that manipulates packed pixels is identical on both big-endian and little-endian systems, you still need to enable byte swapping when drawing packed pixels that have been written in binary form by a system with different endianness.

The Color Matrix Extension

The color matrix extension, SGI_color_matrix, lets you transform the colors in the imaging pipeline with a 4 x 4 matrix. You can use the color matrix to reassign and duplicate color components and to implement simple color-space conversions.

This extension adds a 4 x 4 matrix stack to the pixel transfer path. The matrix operates on RGBA pixel groups, multiplying the 4 x 4 color matrix on top of the stack with the components of each pixel. The stack is manipulated using the OpenGL 1.0 matrix manipulation functions: **glPushMatrix()**, **glPopMatrix()**, **glLoadIdentity()**, **glLoadMatrix()**, and so on. All standard transformations, for example **glRotate()** or **glTranslate()**, also apply to the color matrix.

Below is an example of a color matrix that swaps BGR pixels to form RGB pixels:

```
GLfloat colorMat[16] = {0.0, 0.0, 1.0, 0.0,
                        0.0, 1.0, 0.0, 0.0,
                        1.0, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.0, 0.0 };
glMatrixMode(GL_COLOR);
glPushMatrix();
glLoadMatrixf(colorMat);
```

After the matrix multiplication, each resulting color component is scaled and biased by the appropriate user-defined scale and bias values. Color matrix multiplication follows convolution (and convolution scale and bias).

To set scale and bias values to be applied after the color matrix, call **glPixelTransfer*()** with the following values for *pname:*

- GL_POST_COLOR_MATRIX_{RED/BLUE/GREEN/ALPHA}_SCALE_SGI
- GL_POST_COLOR_MATRIX_{RED/BLUE/GREEN/ALPHA}_BIAS_SGI

The Convolution Extension

The convolution extension, EXT_convolution, allows you to filter images, for example to sharpen or blur them, by convolving the pixel values in a one- or two- dimensional image with a convolution kernel.

The convolution kernels are themselves treated as one- and two- dimensional images. They can be loaded from application memory or from the framebuffer.

Convolution is performed only for RGBA pixel groups, although these groups may have been specified as color indexes and converted to RGBA by index table lookup.

Figure 7-2 shows the equations for general convolution at the top and for separable convolution at the bottom.

$$Dest[i,j] = \sum_{l=0}^{Kh} \sum_{k=0}^{Kw} Kernel[k,l]\, Source[i+k, j+l]$$

$$Dest[i,j] = \sum_{l=0}^{Kh} Vert[l] \sum_{k=0}^{Kw} Horiz[k]\, Source[i+k, l+j]$$

Figure 7-2 Convolution Equations

Performing Convolution

Performing convolution consists of these steps:

1. If desired, specify filter scale, filter bias, and convolution parameters for the convolution kernel. For example:

```
glConvolutionParameteriEXT(GL_CONVOLUTION_2D_EXT,
   GL_CONVOLUTION_BORDER_MODE_EXT,
   GL_REDUCE_EXT /*nothing else supported at present */);
```

```
glConvolutionParameterfvEXT(GL_CONVOLUTION_2D_EXT,
   GL_CONVOLUTION_FILTER_SCALE_EXT,filterscale);
glConvolutionParameterfvEXT(GL_CONVOLUTION_2D_EXT,
   GL_CONVOLUTION_FILTER_BIAS_EXT, filterbias);
```

2. Define the image to be used for the convolution kernel.

Use a 2D array for 2D convolution and a 1D array for 1D convolution. Separable 2D filters consist of two 1D images for the row and the column.

To specify a convolution kernel, call **glConvolutionFilter2DEXT()**, **glConvolutionFilter1DEXT()**, or **glSeparableFilter2DEXT()**.

The following example defines a 7 x 7 convolution kernel that is in RGB format and is based on a 7 x 7 RGB pixel array previously defined as rgbBlurImage7x7:

```
glConvolutionFilter2DEXT(
   GL_CONVOLUTION_2D_EXT,    /*has to be this value*/
   GL_RGB,                   /*filter kernel internal format*/
   7, 7,                     /*width & height of image pixel array*/
   GL_RGB,                   /*image internal format*/
   GL_FLOAT,                 /*type of image pixel data*/
   (const void*)rgbBlurImage7x7      /* image itself*/
   )
```

For more information about the different parameters, see the reference page for the relevant function.

3. Enable convolution.
4. Perform pixel operations (for example pixel drawing or texture image definition).

   Convolution happens as the pixel operations are executed.
5. Use **glGetConvolutionParameter*EXT()** to retrieve the following convolution state parameters:

GL_CONVOLUTION_BORDER_MODE_EXT
: Convolution border mode. For a list of border modes, see **glConvolutionParameterEXT()**.

GL_CONVOLUTION_FORMAT_EXT
: Current internal format. For lists of allowable formats, see **glConvolutionFilter*EXT()**, and **glSeparableFilter2DEXT()**.

GL_CONVOLUTION_FILTER_{BIAS, SCALE}_EXT
: Current filter bias and filter scale factors. *params* must be a pointer to an array of four elements, which receive the red, green, blue, and alpha filter bias terms in that order.

GL_CONVOLUTION_{WIDTH, HEIGHT}_EXT
: Current filter image width.

GL_MAX_CONVOLUTION_{WIDTH, HEIGHT}_EXT
: Maximum acceptable filter image width and filter image height.

Separable and General Convolution Filters

A convolution that uses separable filters operates faster than one that uses general filters.

Special facilities are provided for the definition of two-dimensional separable filters. For separable filters, the image is represented as the product of two one-dimensional images, not as a full two-dimensional image.

To specify a two-dimensional separable filter, call **glSeparableFilter2DEXT()**.

- *target* must be GL_SEPARABLE_2D_EXT.
- *internalformat* specifies the formats of two one-dimensional images that are retained; it must be one of GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_INTENSITY_EXT, GL_RGB, or GL_RGBA.
- *row* and *column* point to two one-dimensional images in memory.
  - The *row* image, defined by *format* and *type*, is *width* pixels wide.
  - The *column* image, defined by *format* and *type*, is *height* pixels wide.

The two images are extracted from memory and processed just as if **glConvolutionFilter1DEXT()** were called separately for each, with the resulting retained images replacing the current 2D separable filter images, except that each scale and bias are applied to each image using the 2D separable scale and bias vectors.

If you're using convolution on a texture image, keep in mind that the result of the convolution must obey the constraint that the dimensions have to be a power of 2. If you use the reduce border convolution mode, the image shrinks by the filter width minus 1, so you may have to take that into account ahead of time.

New Functions

glConvolutionFilter1DEXT(), glConvolutionFilter2DEXT(), glCopyConvolutionFilter1DEXT(), glCopyConvolutionFilter2DEXT(), glGetConvolutionFilterEXT(), glSeparableFilter2DEXT(), glGetSeparableFilterEXT(), glConvolutionParameterivEXT(), glConvolutionParameterfEXT(), glConvolutionParameterfvEXT(), glGetConvolutionParameterivEXT, glGetConvolutionParameterfvEXT().

The Histogram and Minmax Extensions

The histogram extension, EXT_histogram, defines operations that count occurrences of specific color component values and that track the minimum and maximum color component values in images that pass through the image pipeline. You can use the results of these operations to create a more balanced, better-quality image.

Figure 7-3 illustrates how the histogram extension collects information for one of the color components: The histogram has the number of bins specified at creation, and information is then collected about the number of occurrences of that color component. Assuming the example below is for the red component of an image, you can see that R values between 95 and 127 occurred least often and those between 127 and 159 most often.

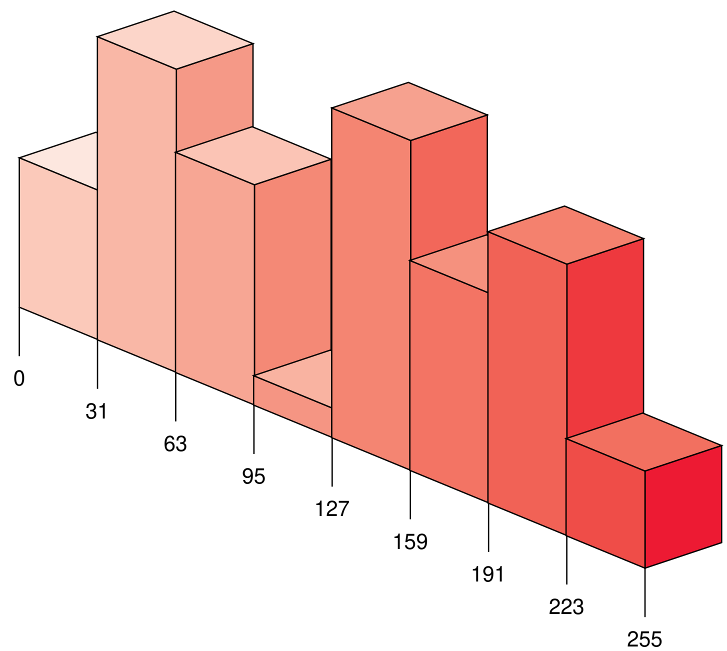


Figure 7-3 How the Histogram Extension Collects Information

Histogram and minmax operations are performed only for RGBA pixel groups, though these groups may have been specified as color indexes and converted to RGBA by color index table lookup.

Using the Histogram Extension

To collect histogram information, follow these steps:

1. Call **glHistogramExt()** to define the histogram, for example:

   ```
   glHistogramEXT(GL_HISTOGRAM_EXT,
                  256               /* width (number of bins) */,
                  GL_LUMINANCE      /* internalformat */,
                  GL_TRUE           /* sink */);
   ```

   - *width*, which must be a power of 2, specifies the number of entries in the histogram.
   - *internalformat* specifies the format of each table entry.
   - *sink* specifies whether pixel groups are consumed by the histogram operation (GL_TRUE) or passed further down the image pipeline (GL_FALSE).

2. Enable histogramming by calling

   ```
   glEnable(GL_HISTOGRAM_EXT)
   ```

3. Perform the pixel operations for which you want to collect information (drawing, reading, or copying pixels).

   For each component represented in the histogram internal format, let the corresponding component of the incoming pixel (luminance corresponds to red) be of value c (after clamping to [0, 1). The corresponding component of bin number `floor((width-1)*c)` is incremented by 1.

4. Call **glGetHistogramEXT()** to query the current contents of the histogram:

   ```
   void glGetHistogramEXT( GLenum target, GLboolean reset, GLenum format,
             GLenum type, GLvoid *values )
   ```

   - *target* must be GL_HISTOGRAM_EXT.
   - *reset* is either GL_TRUE or GL_FALSE. If GL_TRUE, each component counter that is actually returned is reset to zero. Counters that are not returned are not modified.
   - *format* must be one of GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_RGBA, GL_RGB, GL_ABGR_EXT, GL_LUMINANCE, or GL_LUMINANCE_ALPHA.
   - *type* must be GL_UNSIGNED_BYTE, GL_BYTE, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, or GL_FLOAT.

- *values* is used to return a 1-D image with the same width as the histogram. No pixel transfer operations are performed on this image, but pixel storage modes that apply for **glReadPixels()** are performed. Color components that are requested in the specified *format*, but are not included in the internal format of the histogram, are returned as zero. The assignments of internal color components to the components requested by *format* are as follows:

internal component	resulting component
red	red
green	green
blue	blue
alpha	alpha
luminance	red/luminance

Using the Minmax Part of the Histogram Extension

The minmax part of the histogram extension lets you find out about minimum and maximum color component values present in an image. Using the minmax part of the histogram extension is similar to using the histogram part.

To determine minimum and maximum color values used in an image, follow these steps:

1. Specify a minmax table by calling **glMinmaxEXT()**.

   ```
   void glMinmaxEXT( GLenum target, GLenum internalformat, GLboolean sink)
   ```

   - *target* is the table in which the information about the image is to be stored. *target* must be GL_MINMAX_EXT.
   - *internalformat* specifies the format of the table entries. It must be an allowed internal format (see the reference page for **glMinmaxEXT**).
   - *sink* is set to GL_TRUE or GL_FALSE. If GL_TRUE, no further processing happens and pixels or texels are discarded.

   The resulting minmax table always has two entries. Entry 0 is the minimum and entry 1 is the maximum.

2. Enable minmax by calling

   ```
   glEnable(GL_MINMAX_EXT)
   ```

3. Perform the pixel operations, for example, **glCopyPixels()**.

Each component of the internal format of the minmax table is compared to the corresponding component of the incoming RGBA pixel (luminance components are compared to red).

- If a component is greater than the corresponding component in the maximum element, then the maximum element is updated with the pixel component value.
- If a component is smaller than the corresponding component in the minimum element, then the minimum element is updated with the pixel component value.

4. Query the current context of the minmax table by calling **glGetMinMaxExt()**:

```
void glGetMinMaxEXT ( GLenum target, GLboolean reset, GLenum format,
                      GLenum type, glvoid *values)
```

You can also call **glGetMinmaxParameterEXT()** to retrieve minmax state information; setting *target* to GL_MINMAX_EXT and *pname* to one of the following values:

GL_MINMAX_FORMAT_EXT
: internal format of minmax table

GL_MINMAX_SINK_EXT
: value of *sink* parameter

Using Proxy Histograms

Histograms can easily get quite large and require more memory than is available to the graphics subsystem. You can call **glHistogramEXT()** with *target* set to GL_PROXY_HISTOGRAM_EXT to find out whether a histogram fits into memory. The process is similar to that discussed in "The Texture Extension Proxy Mechanism" on page 89 and illustrated in Example 6-1.

To query histogram state values, call **glGetHistogramParameter*EXT()**. Histogram calls with the proxy target (like texture and color table calls with the proxy target) have no effect on the histogram itself.

New Functions

glHistogramEXT(), glResetHistogramEXT(), glGetHistogramEXT(), glGetHistogramParameterivEXT(), glGetHistogramParameterfvEXT() glMinmaxEXT(), glGetMinmaxEXT(), glGetMinmaxParameterivEXT(), glGetMinmaxParametervEXT().

The Color Table Extension

The color table extension, SGI_color_table, defines a new RGBA-format color lookup mechanism. It does not replace the color lookup tables provided by the color maps discussed in the *OpenGL Programming Guide* but provides additional lookup capabilities.

If the copy texture extension is implemented, this extension also defines methods to initialize the color lookup tables from the framebuffer.

Why Use the Color Table Extension?

The color tables provided by the color table extension allow you to adjust image contrast and brightness after each stage of the pixel processing pipeline.

Because you can use several color lookup tables at different stages of the pipeline (see Figure 7-1), you have greater control over the changes you want to make. In addition the extension color lookup tables are more efficient than those of OpenGL 1.0 because you may apply them to a subset of components (for example, Alpha only).

Specifying a Color Table

To specify a color lookup table, call **glColorTableSGI()**:

```
void glColorTableSGI( GLenum target, GLenum internalformat, GLsizei width,
                      GLenum format, GLenum type,const GLvoid *table)
```

- *target* must be GL_COLOR_TABLE_SGI, GL_POST_CONVOLUTION_COLOR_TABLE_SGI, or GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI.
- *internalformat* is the internal format of the color table.

- *width* specifies the number of entries in the color lookup table. It must be zero or a non-negative power of two.
- *format* specifies the format of the pixel data in the table.
- *type* specifies the type of the pixel data in the table.
- *table* is a pointer to a 1D array of pixel data that is processed to build the table.

If no error results from the execution of **glColorTableSGI()**, the following events occur:

1. The specified color lookup table is defined to have *width* entries, each with the specified internal format. The entries are indexed as zero through N-1, where N is the width of the table. The values in the previous color lookup table, if any, are lost. The new values are specified by the contents of the one-dimensional image that *table* points to, with *format* as the memory format and *type* as the data type.
2. The specified image is extracted from memory and processed as if **glDrawPixels()** were called, stopping just before the application of pixel transfer modes (see the illustration "Drawing Pixels with glDrawPixels*()" in Chapter 8, "Bitmaps, Fonts, and Images," of the *OpenGL Programming Guide*).
3. The R, G, B, and A components of each pixel are first scaled by the four GL_COLOR_TABLE_SCALE_SGI parameters, then biased by the four GL_COLOR_TABLE_BIAS_SGI parameters and clamped to [0,1].

   The scale and bias parameters are themselves specified by calling **glColorTableParameterivSGI()** or **glColorTableParameterfvSGI()**:

   - *target* specifies one of the three color tables: GL_COLOR_TABLE_SGI, GL_POST_CONVOLUTION_COLOR_TABLE_SGI, or GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI.
   - *pname* has to be GL_COLOR_TABLE_SCALE_SGI or GL_COLOR_TABLE_BIAS_SGI.
   - *params* points to a vector of four values: red, green, blue, and alpha, in that order.
4. Each pixel is then converted to have the specified internal format. This conversion maps the component values of the pixel (R, G, B, and A) to the values included in the internal format (red, green, blue, alpha, luminance, and intensity).

The new lookup tables are treated as 1-dimensional images with internal formats, like texture images and convolution filter images. As a result, the new tables can operate on a subset of the components of passing pixel groups. For example, a table with internal format GL_ALPHA modifies only the A component of each pixel group, leaving the R, G, and B components unmodified.

Using Framebuffer Image Data for Color Tables

If the copy texture extension is supported, you can define a color table using image data in the framebuffer. Call **glCopyColorTableSGI()**, which accepts image data from a color buffer region (*width* pixel wide by one pixel high) whose left pixel has window coordinates *x,y*. If any pixels within this region are outside the window that is associated with the OpenGL context, the values obtained for those pixels are undefined.

The pixel values are processed exactly as if **glCopyPixels()** had been called, until just before the application of pixel transfer modes (see the illustration "Drawing Pixels with glDrawPixels*()" in Chapter 8, "Bitmaps, Fonts, and Images," of the *OpenGL Programming Guide*).

At this point all pixel component values are treated exactly as if **glColorTableSGI()** had been called, beginning with the scaling of the color components by GL_COLOR_TABLE_SCALE_SGI. The semantics and accepted values of the *target* and *internalformat* parameters are exactly equivalent to their **glColorTableSGI()** counterparts.

Where Are the Lookup Tables in the Image Pipeline?

The three lookup tables exist at different points in the image pipeline (see Figure 7-1):

- GL_COLOR_TABLE_SGI is located immediately after index lookup or RGBA to RGBA mapping, and immediately prior to the convolution operation.
- GL_POST_CONVOLUTION_COLOR_TABLE_SGI is located immediately after the convolution operation (including its scale and bias operations) and immediately prior to the color matrix operation.
- GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI is located immediately after the color matrix operation (including its scale and bias operations) and immediately prior to the histogram operation.

To enable and disable color tables, call **glEnable()** and **glDisable()** with the color table name passed as the *cap* parameter. Color table lookup is performed only for RGBA groups, though these groups may have been specified as color indexes and converted to RGBA by an index-to-RGBA pixel map table.

When enabled, a color lookup table is applied to all RGBA pixel groups, regardless of the command that they are associated with.

New Functions

glColorTableSGI(), glColorTableParameterivSGI(), glGetColorTableSGI(), glGetColorTableParameterivSGI(), glGetColorTableParameterfvSGI().

The Interlace Extension

The interlace extension, SGIX_interlace, provides a way to interlace rows of pixels when rasterizing pixel rectangles, or loading texture images. In this context, interlacing means skipping over rows of pixels or texels in the destination. This is useful for dealing with interlace video data since single frames of video are typically composed of two fields: one field specifies the data for even rows of the frame, the other specifies the data for odd rows of the frame.

Caution: This extension is an SGIX (experimental) extension. The interface may change, or it may be dropped altogether.

Using the Interlace Extension

To turn interlacing on or off, call **glEnable()** or **glDisable()** with the *cap* parameter GL_INTERLACE_SGIX. When enabled, it modifies the behavior of **glDrawPixels()**, **glCopyPixels()**, **glTexImage2D()**, **glTexSubImage2DEXT()**, **glCopyTexImage2DEXT()** and **glCopyTexSubImage2DEXT()** as well as the 3D versions of texture calls.

If GL_INTERLACE_SGIX is enabled, all the groups that belong to a row m are treated as if they belonged to the row $2 \times m$. If the source image has a height of h rows, this effectively expands the height of the image to $2 \times h$ rows.

After interlacing, only every other row of the image is defined:

- If the interlaced pixel rectangle is rasterized to the framebuffer, then only those rows are converted to fragments.

- If the interlaced pixel rectangle is a texture image, then only those rows are written to texture memory.

In cases where errors can result from the specification of invalid image dimensions, the resulting dimensions—not the dimensions of the source image—are tested.

For example, when you use **glTexImage2D()** with GL_INTERLACE_SGIX enabled, the source image you provide must be of height `(texture_height + texture_border)/2`.

How to Use Interlace and Copy Texture Together

One application of the interlace extension is to use it together with the copy texture extension: You can use **glCopyTexSubImage2DEXT()** to copy the contents of the video field to texture memory and end up with de-interlaced video.

You can interlace pixels from two images as follows:

```
glEnable(GL_INTERLACE_SGIX);
< set current raster position to (xr,yr) >
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I0);
< set raster position to (xr,yr+zoomy) >
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I1);
```

This process is equivalent to taking pixel rows (0,2,4,...) of I2 are from image I0, and rows (1,3,5,...) from image I1, as follows:

```
glDisable( GL_INTERLACE_SGIX);
< set current raster position to (xr,yr)>
glDrawPixels(width, 2*height, GL_RGBA, GL_UNSIGNED_BYTE, I2);
```

Blending Extensions

Blending refers to the process of combining color values from a source (an incoming pixel fragment) with current values of the stored pixel in the framebuffer (the destination). The final effect is that parts of a scene appear translucent. In OpenGL 1.0, you specify the blending operation by calling **glBlendFunc()**, then enable or disable blending using **glEnable()** or **glDisable()** with GL_BLEND.

Blending is discussed in the first section of Chapter 7, “Blending, Antialiasing, and Fog,” of the *OpenGL Programming Guide*. The section also lists a number of sample uses of blending.

This section explains how to use extensions that support color blending for images and rendered geometry in a variety of ways:

- “The Constant Color Blending Extension”
- “The Minmax Blending Extension”
- “The Logical Operation Blending Extension”
- “The Blend Subtract Extension”

The Constant Color Blending Extension

The standard blending feature allows you to blend source and destination pixels. The constant color blending extension, EXT_blend_color, enhances this capability by defining a constant color that you can include in blending equations.

Constant color blending allows you to specify input source with constant alpha that is not 1 without actually specifying the alpha for each pixel. Alternatively, when working with visuals that have no alpha, you can use the blend color for constant alpha. This also allows you to modify a whole incoming source by blending with a constant color (which is faster than clearing to that color). In effect, the image looks as if it is viewed through colored glasses.

Using Constant Colors for Blending

To use a constant color for blending, follow these steps:

1. Call **glBlendColorEXT()** to specify the blending color:

   ```
   void glBlendColorEXT( GLclampf red, GLclampf green, GLclampf blue,
                         GLclampf alpha )
   ```

   The four parameters are clamped to the range [0,1] before being stored. The default value for the constant blending color is (0,0,0,0).

2. Call **glBlendFunc()** to specify the blending function, using one of the tokens listed in Table 7-2 as source or destination factor, or both.

Table 7-2 Blending Factors Defined by the Blend Color Extension

Constant	Computed Blend Factor
GL_CONSTANT_COLOR_EXT	(Rc, Gc, Bc, Ac)
GL_ONE_MINUS_CONSTANT_COLOR_EXT	(1, 1, 1, 1) - (Rc, Gc, Bc, Ac)
GL_CONSTANT_ALPHA_EXT	(Ac, Ac, Ac, Ac)
GL_ONE_MINUS_CONSTANT_ALPHA_EXT	(1, 1, 1, 1) - (Ac, Ac, Ac, Ac)

Rc, Gc, Bc, and Ac are the four components of the constant blending color. These blend factors are already in the range [0,1].

You can, for example, fade between two images by drawing both images with Alpha and 1-Alpha as Alpha goes from 1 to 0, as in the following code fragment:

```
glBlendFunc(GL_ONE_MINUS_CONSTANT_COLOR_EXT, GL_CONSTANT_COLOR_EXT);
for (alpha = 0.0; alpha <= 1.0; alpha += 1.0/16.0) {
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, image0);
    glEnable(GL_BLEND);
    glBlendColorEXT(alpha, alpha, alpha, alpha);
    glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, image1);
    glDisable(GL_BLEND);
    glXSwapBuffers(display, window);
    }
```

New Functions

glBlendColorEXT().

The Minmax Blending Extension

The minmax blending extension, EXT_blend_minmax, extends blending capability by introducing two new equations that produce the minimum or maximum color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

This extension also introduces a mechanism for defining alternate blend equations. Note that even if the minmax blending extension is not supported on a given system, that system may still support the logical operation blending extension or the subtract blending extension. When these extensions are supported, the **glBlendEquationEXT()** function is also supported.

Using a Blend Equation

To specify a blend equation, call **glBlendEquationEXT()**:

```
void glBlendEquationEXT(GLenum mode)
```

The *mode* parameter specifies how source and destination colors are combined. The blend equations GL_MIN_EXT, GL_MAX_EXT, and GL_LOGIC_OP_EXT use source or destination factors.

If *mode* is set to GL_FUNC_ADD_EXT, then the blend equation is set to GL_ADD, the equation used currently in OpenGL 1.0. The **glBlendEquationEXT()** reference page lists other modes. These modes are also discussed in "The Blend Subtract Extension" on page 144 and "The Logical Operation Blending Extension" on page 143. While OpenGL 1.0 defines logic operation only on color indices, this extension extends the logic operation to RGBA pixel groups. The operation is applied to each component separately.

New Functions

glBlendEquationEXT()

The Logical Operation Blending Extension

The logical operation blending extension, EXT_blend_logic_op, specifies a single additional blending equation, using the **glBlendEquationEXT()** call discussed in "The Minmax Blending Extension" on page 142. The blending equation is a logical combination of the source and destination colors. The specific logical operation is specified by **glLogicOp()**, which is discussed in the section "Logical Operations" in Chapter 10, "The Framebuffer" of the *OpenGL Programming Guide.*

A possible application of this extension is for annotations, for example, assume you want to annotate a map without redrawing the map itself. To make the annotations visible, XOR them with the map to make them disappear again, XOR them once more. You should XOR the most significant bit of the color.

If you call **glBlendEquationEXT()** with *mode* set to GL_LOGIC_OP, the blending equation becomes

```
C = Cs OP Cd
```

Cs and Cd are the source and destination colors, and OP is the logic operation as specified by **glLogicOp()**. The value of the boolean parameter GL_LOGIC_OP, specified by **glEnable()** and **glDisable()**, has no effect on blending.

The Blend Subtract Extension

The blend subtract extension, EXT_blend_subtract, provides two additional blending equations that can be used by **glBlendEquationEXT()**. These equations are similar to the default blending equation, but produce the difference of its left- and right-hand sides, rather than the sum. See the reference page for **glBlendEquationEXT()** for a detailed description.

Image differences are useful in many image-processing applications; for example, comparing two pictures that may have changed over time.

Miscellaneous OpenGL Extensions

This chapter explains how to use several extensions that are not easily grouped with texturing, imaging, or GLX extensions, providing example code as needed. You learn about:

The Polygon Offset Extension

The polygon offset extension, EXT_polygon_offset, is useful for rendering hidden-line images, rendering solids with highlighted edges, and applying decals to surfaces.

The extension is useful, for example, when you want to render outlines on surfaces. The lines are not rasterized the same way the polygons are. This results in slightly different depth values which, when used by the depth buffering mechanism, cause some of the line pixels to show, and some not to show. This is because triangles and lines produce different depth values and the system cannot tell ahead of time what will be in front. The extension offsets polygon pixels in the depth buffer so that lines and points no longer interact in that way, resulting in clean polygon outlines.

Using the extension, you can displace the depth values of fragments generated by rendering polygons by a fixed bias plus an amount. The amount is proportional to the maximum absolute value of the depth slope of the polygon, measured and applied in window coordinates. This displacement

- allows lines (or points) and polygons in the same plane to be rendered without interaction: the lines are rendered either completely in front of or behind the polygons (depending on the sign of the offset factor)
- allows multiple coplanar polygons to be rendered without interaction, if different offset factors are used for each polygon

Polygon Offset Example Program

This section illustrates how to use the polygon offset extension by providing code fragments from an example program *hiddenline.c* that displays three images:

- The left image shows an ordinary wireframe image.
- The center image shows a hidden-line image without polygon offset.
- The right image shows a hidden-line image using polygon offset to reduce artifacts.

Figure 8-1 shows the output that the program generates. You are encouraged to run the program yourself to see the difference more clearly than the printed version can show.

Example 8-1 Polygon Offset Example Program

```
/**********************************************************************
 * Uses PolygonOffset to draw hidden-line images. PolygonOffset
 * shifts z values of polygons by an amount proportional to their slope
 * in screen z. This keeps the lines, which are drawn without
 * displacement, from interacting with their respective polygons, and
 * and thus eliminates line dropouts.
 */

#include <GL/glx.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAXQUAD 6

typedef float Vertex[3];

typedef Vertex Quad[4];

Quad quads[MAXQUAD] = {    /*data to define six faces of a unit cube */
    0,0,0, 1,0,0, 1,1,0, 0,1,0,
    0,0,1, 1,0,1, 1,1,1, 0,1,1,
    0,0,0, 1,0,0, 1,0,1, 0,0,1,
    0,1,0, 1,1,0, 1,1,1, 0,1,1,
    0,0,0, 0,0,1, 0,1,1, 0,1,0,
    1,0,0, 1,0,1, 1,1,1, 1,1,0
};
```

```
#define WIREFRAME       0
#define HIDDEN_LINE     1

/*** function prototypes go here  ***/
static int dimension = 3;

main(int argc, char** argv) {
    Display *dpy;
    XVisualInfo *vi;
    XSetWindowAttributes swa;
    Window win;
    GLXContext cx;

    /*** X Window System setup goes here ***/

    /* check for the polygon offset extension */
    if (!query_extension("GL_EXT_polygon_offset"))
        error(argv[0], "polygon_offset extension is not available");

    /* set up viewing parameters */
    glMatrixMode(GL_PROJECTION);
    gluPerspective(20, 1, 0.1, 20);
    glMatrixMode(GL_MODELVIEW);
    glTranslatef(0, 0, -15);

    /* set other relevant state information */
    glEnable(GL_DEPTH_TEST);
    glPolygonOffsetEXT(1.5, 0.000001);

    /* process events until the user presses ESC */
    while (1) process_input(dpy, win);
}

static void
draw_scene(int mx, int my) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glTranslatef(-1.7, 0.0, 0.0);
    cubes(mx, my, WIREFRAME);
    glPopMatrix();

    glPushMatrix();
    cubes(mx, my, HIDDEN_LINE);
    glPopMatrix();
```

```
    glPushMatrix();
    glTranslatef(1.7, 0.0, 0.0);
    glEnable(GL_POLYGON_OFFSET_EXT);
    cubes(mx, my, HIDDEN_LINE);
    glDisable(GL_POLYGON_OFFSET_EXT);
    glPopMatrix();
}

static void
cubes(int mx, int my, int mode) {
    int x, y, z, i;

    /* track the mouse */
    glRotatef(mx / 2.0, 0, 1, 0);
    glRotatef(my / 2.0, 1, 0, 0);

    /* draw the lines as hidden polygons */
    glTranslatef(-0.5, -0.5, -0.5);
    glScalef(1.0/dimension, 1.0/dimension, 1.0/dimension);
    for (z = 0; z < dimension; z++) {
        for (y = 0; y < dimension; y++) {
            for (x = 0; x < dimension; x++) {
                glPushMatrix();
                glTranslatef(x, y, z);
                glScalef(0.8, 0.8, 0.8);
                for (i = 0; i < MAXQUAD; i++)
                    draw_hidden(quads[i], mode);
                glPopMatrix();
            }
        }
    }
}

static void
fill(Quad quad) {
    glBegin(GL_QUADS);
    glVertex3fv(quad[0]);
    glVertex3fv(quad[1]);
    glVertex3fv(quad[2]);
    glVertex3fv(quad[3]);
    glEnd();
}
```

```
static void
outline(Quad quad) {
    /* draw an outlined polygon */
    glBegin(GL_LINE_LOOP);
    glVertex3fv(quad[0]);
    glVertex3fv(quad[1]);
    glVertex3fv(quad[2]);
    glVertex3fv(quad[3]);
    glEnd();
}

static void
draw_hidden(Quad quad, int mode) {
    /* draw outline using white, optionally fill interior with black */
    glColor3f(1, 1, 1);
    outline(quad);

    if (mode == HIDDEN_LINE) {
        glColor3f(0, 0, 0);
        fill(quad);
    }
}

static void
process_input(Display *dpy, Window win) {
    XEvent event;
    static int prevx, prevy;
    static int deltax = 90, deltay = 40;

    do {
        char buf[31];
        KeySym keysym;

        XNextEvent(dpy, &event);
        switch(event.type) {
        case Expose:
            break;
        case ConfigureNotify:
            glViewport(0, 0, e+vent.xconfigurewidth,
                       event.xconfigureheigth);
            break;
        case ButtonPress:
            prevx = event.xbutton.x;
            prevy = event.xbutton.y;
            break;
```

```
            case MotionNotify:
                deltax += (event.xbutton.x - prevx); prevx =
                           event.xbutton.x;
                deltay += (event.xbutton.y - prevy); prevy =
                           event.xbutton.y;
                break;
            default:
                break;
            }
        } while (XPending(dpy));

        draw_scene(deltax, deltay);
        glXSwapBuffers(dpy, win);
    }
```

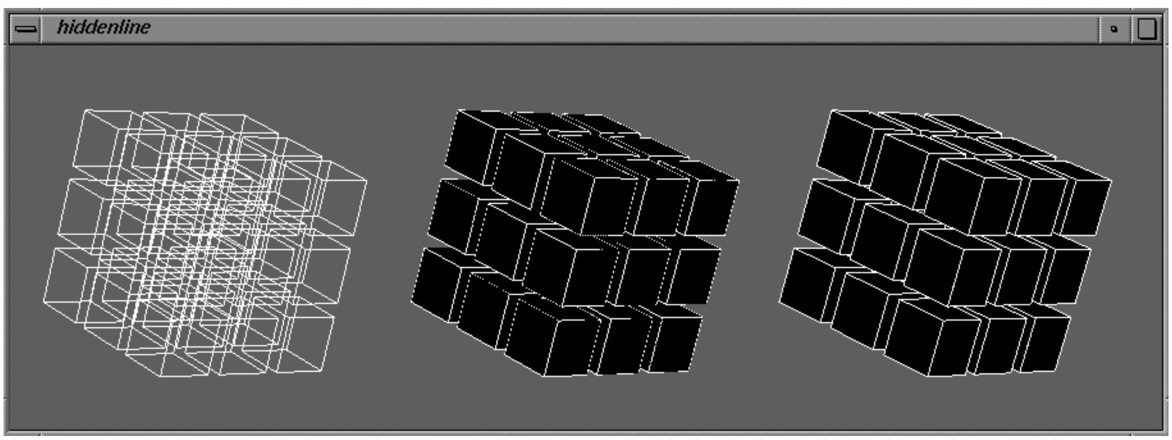


Figure 8-1 Polygon Offset Extension Example

New Functions

glPolygonOffsetEXT().

The Vertex Array Extension

The vertex array extension, EXT_vertex_array, makes it possible to specify multiple geometric primitives with very few subroutine calls. This section first discusses why and when you might want to use the extension, then looks at how the extension works, including a discussion of static and nonstatic arrays, and how to use it.

Why and When to Use the Vertex Array Extension

Instead of calling an OpenGL procedure to pass each individual vertex, normal, or color, you can specify separate arrays of vertexes, normals, and colors, and use them to define a sequence of primitives (all of the same type) when a single call is made to **glDrawArraysEXT()**.

The extension defines a stride mechanism that lets an application choose to keep all data staggered in a single array. For example, you could combine color, vertex, and edge data and access each type as needed using the stride number. This extension also supports the rendering of individual array elements, each specified as an index into the enabled arrays.

Note: Using the vertex array extension may lead to performance improvements or performance deterioration. While using the extension reduces the number of function calls your application has to make, there is added overhead for accessing the data in the array. The break-even point differs depending on the system the application is running on. See “Optimizing Use of the Vertex Array Extension” on page 284 for information on the accelerated cases on RealityEngine systems.

On Silicon Graphics systems, single-array storage provides better performance. The extension has been optimized for floats but not for other data types. You should run some performance tests to determine whether using vertex arrays is really appropriate for your application.

How the Vertex Array Extension Works

The extension maintains individual array pointers and associated data for an array of elements: vertexes, normals, colors, color indexes, texture coordinates, and edge flags. The data associated with each array specify the following:

- the data type of the values in the array

- the number of values per element in the array (for example, vertexes of 2, 3, or 4 coordinates)
- the byte stride from one array element to the next
- the number of elements (counting from the first) that are static

  Static elements may be modified by the application, but once they are modified, the application must explicitly respecify the array before using it for rendering.

When you specify an array, the pointer and associated data are saved as client-side state, and static elements may be cached. You define an element as static, that is, guaranteed not to change, using the *count* argument to **glVertexPointerEXT().** Non-static (dynamic) elements are never accessed until a call to **glArrayElementEXT()** or **glDrawArraysEXT().**

Specifying Arrays for Different Kinds of Data

This section provides information about specifying arrays for vertices. A full description for normals, colors, color indexes, texture coordinates, and edge flags is not included; see the reference pages for more information.

- **glVertexPointerEXT()** specifies the location and data format of an array of vertex coordinates.
  – *pointer* specifies a pointer to the first coordinate of the first vertex in the array.
  – *type* specifies the data type of each coordinate in the array, and must be one of GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE_EXT, implying OpenGL data types short, int, float, and double, respectively.
  – *size* specifies the number of coordinates per vertex, and must be 2, 3, or 4.
  – *stride* specifies the byte offset between pointers to consecutive vertexes. If *stride* is zero, the vertex data are tightly packed in the array.
  – *count* specifies the number of vertexes, counting from the first, that are static.
- **glNormalPointerEXT()** specifies the location and data format of an array of normals.
- **glColorPointerEXT()** specifies the location and data format of an array of color components.
- **glIndexPointerEXT()** specifies the location and data format of an array of color indexes.

- **glTexCoordPointerEXT()** specifies the location and data format of an array of texture coordinates.
- **glEdgeFlagPointerEXT()** specifies the location and data format of an array of boolean edge flags.

Rendering the Arrays

By default all the arrays are disabled; they are not accessed when either **glArrayElementEXT()** or **glDrawArraysEXT()** is called. To enable or disable an individual array, call **glEnable()** or **glDisable()** with *cap* set to an appropriate value, as specified in Table 8-1.

Table 8-1 Tokens for Enabling Arrays

Array Specification Command	Enable Token
glVertexPointerEXT	GL_VERTEX_ARRAY_EXT
glNormalPointerEXT	GL_NORMAL_ARRAY_EXT
glColorPointerEXT	GL_COLOR_ARRAY_EXT
glIndexPointerEXT	GL_INDEX_ARRAY_EXT
glTexCoordPointerEXT	GL_TEXTURE_COORD_ARRAY_EXT
glEdgeFlagPointerEXT	GL_EDGE_FLAG_ARRAY_EXT

When **glArrayElementEXT()** is called, a single vertex is drawn, using vertex and attribute data taken from location *i* of the enabled arrays.

Even if GL_VERTEX_ARRAY_EXT is not enabled, the **glArrayElementEXT()** function executes. No drawing occurs in this case, but the attributes corresponding to enabled arrays are modified.

When **glDrawArraysEXT()** is called, *count* sequential elements from each enabled array are used to construct a sequence of geometric primitives, beginning with element *first*. The *mode* parameter specifies what kind of primitives are constructed, and how the array elements are used to construct these primitives. Accepted values for *mode* are GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_QUAD_STRIP, GL_QUADS, and

GL_POLYGON. If GL_VERTEX_ARRAY_EXT is not enabled, no geometric primitives are generated.

Vertex attributes that are modified by **glDrawArraysEXT()** have an unspecified value after **glDrawArraysEXT()** returns. For example, if GL_COLOR_ARRAY_EXT is enabled, the value of the current color is undefined after **glDrawArraysEXT()** executes. Attributes that aren't modified remain well defined.

If an error is generated during execution of **glDrawArraysEXT()**, no other operations take place and the state does not change.

Although it is not an error to respecify an array between the execution of **glBegin()** and the corresponding execution of **glEnd()**, the result of such respecification is undefined. Static array data may be read and cached by the implementation at any time. If static array data are modified by the application, the results of any subsequent **glArrayElementEXT()** or **glDrawArraysEXT()** calls are undefined.

New Functions

glArrayElementEXT(), glDrawArraysEXT(), glVertexPointerEXT(), glNormalPointerEXT(), glColorPointerEXT(), glIndexPointerEXT(), glTexCoordPointerEXT(), glEdgeFlagPointerEXT(), glGetPointervEXT().

The Multisample Extension

The multisample extension, SGIS_multisample, provides a mechanism to antialias all OpenGL primitives: points, lines, polygons, bitmaps, and images.

This section explains how to use multisampling and also discusses what happens when you use it, discussing the following topics:

- "Introduction to Multisampling" on page 155
- "Using the Multisample Extension" on page 155 and "Using Advanced Multisampling Options" on page 156
- "How Multisampling Affects Different Primitives" on page 160

Introduction to Multisampling

Multisampling works by sampling all primitives multiple times at different locations within each pixel, in effect collecting subpixel information. The result is an image that has fewer aliasing artifacts.

Because each sample includes depth and stencil information, the depth and stencil functions perform equivalently to the single-sample mode. A single pixel can have 4, 8, or 16 subsamples.

When you use multisampling and read back color, you get the resolved color value (that is, the average of the samples). When you read back stencil or depth, you typically get back a single sample value rather than the average; the sample value is typically the one closest to the center of the pixel.

When to Use Multisampling

Multisample antialiasing is most valuable for rendering polygons because it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. Each time a pixel is updated, the color sample values for each pixel are resolved to a single, displayable color.

For points and lines, the "smooth" antialiasing mechanism provided by standard OpenGL results in a higher-quality image and should be used instead (see "Antialiasing" in Chapter 7, "Blending, Antialiasing, and Fog," of the *OpenGL Programming Guide).*

The multisampling extension lets you alternate multisample and smooth antialiasing during the rendering of a single scene, so it's possible to mix multisampled polygons with smooth lines and points. See "Multisampled Points" on page 160 and "Multisampled Lines" on page 160 for more information.

Using the Multisample Extension

To use multisampling in your application, select a multisampling-capable visual by calling **glXChooseVisual()** with the following items in *attr_list*:

GLX_SAMPLES_SGIS

Must be followed by the minimum number of samples required in multisample buffers. **glXChooseVisual()** gives preference to visuals with the smallest number of samples that meet or exceed the specified number. Color samples in the multisample buffer may have fewer bits than colors in the main color buffers. However, multisampled colors maintain at least as much color resolution in aggregate as the main color buffers.

GLX_SAMPLE_BUFFERS_SGIS
: This attribute is optional. Currently there are no visuals with more than one multisample buffer, so the returned value is either zero or one. When GLX_SAMPLES_SGIS is non-zero, this attribute defaults to 1. When specified, the attribute must be followed by the minimum acceptable number of multisample buffers. Visuals with the smallest number of multisample buffers that meet or exceed this minimum number are preferred.

Multisampling is enabled by default.

- To query whether multisampling is enabled, call

  ```
  glIsEnabled(MULTISAMPLE_SGIS)
  ```
- To turn off multisampling, call

  ```
  glDisable(MULTISAMPLE_SGIS)
  ```

Using Advanced Multisampling Options

Advanced multisampling options provide additional rendering capabilities. This section discusses:

- using a multisample mask to choose how many samples are writeable
- using alpha values to feather-blend texture edges
- using the accumulation buffer with multisampling

Figure 8-2 shows how the subsamples in one pixel are turned on and off.

1. First, the primitive is sampled at the locations defined by a sample pattern. If a sample is inside the polygon, it is turned on, otherwise, it is turned off. This produces a coverage mask.
2. The coverage mask is then ANDed with a user-defined sample mask, defined by a call to **glSampleMaskSGIS() (**see “Using a Multisample Mask to Fade Levels of Detail” on page 158).

3. You may also choose to convert the alpha value of a fragment to a mask and AND it with the coverage mask from step 2.

   Enable GL_SAMPLE_ALPHA_TO_MASK_SGIS to convert alpha to the mask. The fragment alpha value is used to generate a temporary mask, which is then ANDed with the fragment mask.

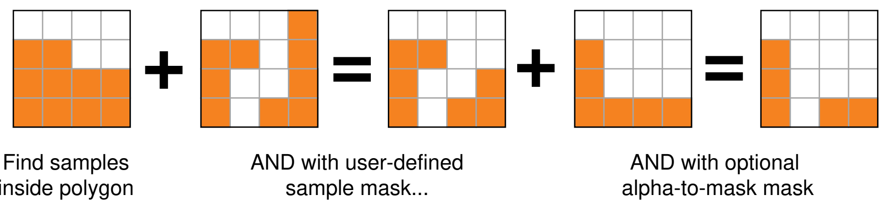


Figure 8-2 Sample Processing During Multisampling

The two processes—using a multisample mask created by **glSampleMaskSGIS()** and using the alpha value of the fragment as a mask—can both be used for different effects.

When GL_SAMPLE_ALPHA_TO_MASK_SGIS is enabled, it's usually appropriate to enable GL_SAMPLE_ALPHA_TO_ONE_SGIS to convert the alpha values to 1 before blending. Without this option, the effect would be colors that are twice as transparent.

Note: When you use multisampling, blending reduces performance. Therefore, when possible, disable blending and instead use GL_SAMPLE_MASK_SGIS or GL_ALPHA_TO_MASK.

Color Blending and Screen-Door Transparency

Multisampling can be used to solve the problem of blurred edges on textures with irregular edges, such as tree textures, that require extreme magnification. When the texture is magnified, the edges of the tree look artificial, as if the tree were a paper cutout. To make them look more natural by converting the alpha to a multisample mask, you can obtain several renderings of the same primitive, each with the samples offset by a specific amount. See "Accumulating Multisampled Images" on page 159 for more information.

The same process can be used to achieve screen-door transparency: If you draw only every other sample, the background shines through for all other samples, resulting in a transparent image. This is useful because it doesn't require the polygons to be sorted from back to front. It is also faster because it doesn't require blending.

Using a Multisample Mask to Fade Levels of Detail

You can use a mask to specify a subset of multisample locations to be written at a pixel. This feature is useful for implementing fade-level-of-detail in visual simulation applications. You can use multisample masks to perform the blending from one level of detail of a model to the next by rendering the additional data in the detailed model using a steadily increasing percentage of subsamples as the viewpoint nears the object.

To achieve this blending between a simpler and a more detailed representation of an object, or to achieve screen-door transparency discussed in the previous section, either call **glSampleMaskSGIS()** or use the Alpha values of the object and call **glSampleAlphaToMaskSGIS().** Below is the prototype for **glSampleMaskSGIS()**:

```
void glSampleMaskSGIS (GLclampf value, boolean invert);
```

- *value* specifies coverage of the modification mask. Clamped to the range [0, 1]; 0 implies no coverage, and 1 implies full coverage.
- *invert* should be GL_FALSE to use the modification mask implied by value or GL_TRUE to use the bitwise inverse of that mask.

To define a multisample mask using **glSampleMaskSGIS()**, follow these steps:

1. Enable GL_SAMPLE_MASK_SGIS.
2. Call **glSampleMaskSGIS()** with, for example, *value* set to .25 and *invert* set to GL_FALSE.
3. Render the object once for the more complex level of detail.
4. Call **glSampleMaskSGIS()** again with, for example, *value* set to .25 and *invert* set to GL_TRUE.
5. Render the object for the simpler level of detail.

   This time, the complementary set of samples is used because of the use of the inverted mask.
6. Display the image.
7. Repeat the process for larger sample mask values of the mask as needed (as the viewpoint nears the object).

Accumulating Multisampled Images

You can enhance the quality of the image even more by making several passes, adding the result in the accumulation buffer. The accumulation buffer averages several renderings of the same primitive. For multipass rendering, different sample locations need to be used in each pass to achieve high quality.

When an application uses multisampling in conjunction with accumulation, it has to call **glSamplePatternSGIS()** with one of the following patterns as an argument:

- GL_1PASS_SGIS is designed to produce a well-antialiased result in a single rendering pass (this is the default).
- GL_2PASS_0_SGIS and GL_2PASS_1_SGIS together specify twice the number of sample points per pixel. You should first completely render a scene using pattern GL_2PASS_0_SGIS, then completely render it again using GL_2PASS_1_SGIS. When the two images are averaged using the accumulation buffer, the result is as if a single pass had been rendered with 2×GL_SAMPLES_SGIS sample points.
- GL_4PASS_0_SGIS, GL_4PASS_1_SGIS, GL_4PASS_2_SGIS, and GL_4PASS_3_SGIS together define a pattern of 4×GL_SAMPLES_SGIS sample points. They can be used to accumulate an image from four complete rendering passes.

Accumulating multisample results can also extend the capabilities of your system. For example, if you have only enough resources to allow four subsamples, but you are willing to render the image twice, you can achieve the same effect as multisampling with eight subsamples. Note that you do need an accumulation buffer, which also takes space.

To query the sample pattern, call **glGetIntegerv()** with *pname* set to GL_SAMPLE_PATTERN_SGIS. The pattern should be changed only between complete rendering passes.

For more information, see “The Accumulation Buffer,” in Chapter 10, “The Framebuffer,” of the *OpenGL Programming Guide.*

How Multisampling Affects Different Primitives

This section briefly discusses multisampled points, lines, polygons, pixels, and bitmaps.

Multisampled Points

If you're using multisampling, the value of the smoothing hint (GL_POINT_SMOOTH_HINT or GL_LINE_SMOOTH_HINT) is ignored. Since the quality of multisampled points may not be as good as that of anti-aliased points, remember that you can turn multisampling on and off as needed to achieve multisampled polygons and anti-aliased points.

Note: On RealityEngine systems, you achieve higher-quality multisampled points by setting point smooth hint GL_NICEST (though this mode is slower and should be used with care).

```
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)
```

The result is round points. Points may disappear or flicker if you use them without this hint. See the **Caution:** in the next section for caveats on using multisampling with smooth points.

Multisampled Lines

Lines are sampled into the multisample buffer as rectangles centered on the exact zero-area segment. Rectangle width is equal to the current linewidth. Rectangle length is exactly equal to the length of the segment. The rectangles of colinear, abutting line segments abut exactly, so no subsamples are missed or drawn twice near the shared vertex.

Just like points, lines on RealityEngine systems look better when drawn "smooth" than they do with multisampling.

Caution: If you want to draw smooth lines and points by enabling GL_LINE_SMOOTH_HINT or GL_POINT_SMOOTH_HINT, you need to disable multisampling and then draw the lines and points. The trick is that you need to do this after you have finished doing all of the multisampled drawing. If you try to re-enable multisampling and draw more polygons, those polygons are not necessarily anti-aliased correctly.

Multisampled Polygons

Polygons are sampled into the multisample buffer much as they are into the standard single-sample buffer. A single color value is computed for the entire pixel, regardless of the number of subsamples at that pixel. Each sample is then written with this color if and only if it is geometrically within the exact polygon boundary.

If the depth-buffer is enabled, the correct depth value at each multisample location is computed and used to determine whether that sample should be written or not. If stencil is enabled, the test is performed for each sample.

Polygon bit patterns apply equally to all sample locations at a pixel. All sample locations are considered for modification if the pattern bit is 1. None is considered if the pattern bit is 0.

Multisample Rasterization of Pixels and Bitmaps

In IRIS GL, pixels are not multisampled. In OpenGL, if multisampling is on, pixels are considered small rectangles and are subject to multisampling.

When pixels are sampled into the multisample buffer, each pixel is treated as an xzoom-by-yzoom square, which is then sampled just like a polygon.

For information about fast clears on RealityEngine, see the reference page for **glTagSampleBufferSGIX().**

New Functions

glSampleMaskSGIS(), glSamplePatternSGIS().

Chapter 9

Extensions to GLX

GLX is an extension to the X Window System that makes OpenGL available in an X Window System environment. All GLX functions and other elements have the prefix **glX** (just as all OpenGL elements have the prefix **gl**) For more information about GLX, see "The GLX Extension to the X Window System" on page 2 in this document, and Appendix D, "The OpenGL Extension to the X Window System," in the *OpenGL Programming Guide*.

This chapter explains how to use extensions to GLX. You learn about:

- "The Make Current Read Extension" on page 164
- "The Visual Info Extension" on page 165
- "The Visual Rating Extension" on page 167
- "The Video Synchronization Extension" on page 167
- "The Swap Control Extension" on page 168
- "The Import Context Extension" on page 169

The following sections describe extensions that are experimental:

- "The Framebuffer Configuration Extension" on page 171
- "The Pixel Buffer Extension" on page 179
- "The Video Source Extension" on page 183

Note: Using OpenGL in an X Window System environment is discussed in the following chapters of this guide:

- Chapter 2, "Using OpenGL and X: Getting Started"
- Chapter 3, "OpenGL and X: Examples"
- Chapter 4, "OpenGL and X: Advanced Topics"

The Make Current Read Extension

The make current read extension, SGI_make_current_read, allows you to attach separate read and write drawables to a GLX context by calling **glXMakeCurrentReadSGI()**.

In GLX 1.1, you associate a GLX context with one drawable (window or pixmap) calling **glXMakeCurrent()**. **glXMakeCurrentReadSGI()** lets you attach a GLX context to two drawables: The first is the one you draw to, the second serves as a source for pixel data.

Having both a read and write drawable is useful under various circumstances; for example, to copy the contents of a window to another window, to stream video to a window, and so on.

The write drawable is used for all OpenGL operations. Accumulation buffer operations fetch data from the write drawable and are not allowed when the read and write drawable are not identical.

The read drawable is used for any color, depth, or stencil values that are retrieved by **glReadPixels()**, **glCopyPixels()**, or any OpenGL extension that sources depth images from the framebuffer in the manner of **glReadPixels()** and **glCopyPixels()**.

Here's some additional information about the two drawables:

- The two drawables don't need to have the same ancillary buffers.
- The read drawable doesn't have to contain a buffer corresponding to the current GL_READ_BUFFER of a GLX context. For example, the current GL_READ_BUFFER may be GL_BACK, and the read drawable may be single-buffered.

  If a subsequent command sets the read buffer to a color buffer that doesn't exist on the read drawable—even if set implicitly by **glPopAttrib()**—or if an attempt is made to source pixel values from an unsupported ancillary buffer, a GL_INVALID_OPERATION error is generated.
- If the current GL_READ_BUFFER does not exist in the read drawable, pixel values extracted from that drawable are undefined, but no error is generated.
- Operations that query the value of GL_READ_BUFFER use the value set last in the context, regardless of whether the read drawable has the corresponding buffer.

Possible Match Errors

When **glXMakeCurrentReadSGI()** associates two GLX drawables with a single GLX context, a BadMatch X protocol error is generated if either drawable was not created with the same X screen and visual as the context.

The color, depth, stencil, and accumulation buffers of the two drawables don't need to match. Certain implementations may impose additional constraints. For example, the current RealityEngine implementation requires that the color component resolution of both drawables is the same.

New Functions

glXMakeCurrentReadSGI(), glXGetCurrentReadDrawableSGI().

The Visual Info Extension

The visual info extension, EXT_visual_info, enhances the standard GLX visual mechanism as follows:

- You can request that a particular X visual type be associated with a GLX visual.
- You can query the X visual type underlying a GLX visual.
- You can request a visual with a transparent pixel.
- You can query whether a visual supports a transparent pixel value and query the value of the transparent pixel.

  Note that the notions of level and transparent pixels are orthogonal as both level 1 and level 0 visuals may or may not support transparent pixels.

Using the Visual Info Extension

To find a visual that best matches specified attributes, call **glXChooseVisual()**:

`XVisualInfo* glXChooseVisual( Display *dpy, int screen, int *attrib_list )`

- If GLX_RGBA is in *attrib_list*, you can choose a TrueColor or DirectColor visual by setting GLX_X_VISUAL_TYPE_EXT to GLX_TRUE_COLOR_EXT or GLX_DIRECT_COLOR_EXT, respectively.

  - If you set GLX_X_VISUAL_TYPE_EXT to PseudoColor, StaticColor, GrayScale, or StaticGray, visual selection will fail.
  - If GLX_X_VISUAL_TYPE_EXT is not in *attrib_list*, and if all other attributes are equivalent, a TrueColor visual is chosen in preference to a DirectColor visual.
- If GLX_RGBA is not in *attrib_list*, you can choose a PseudoColor or StaticColor visual by setting GLX_X_VISUAL_TYPE_EXT to GLX_PSEUDO_COLOR or GLX_STATIC_COLOR, respectively.
  - If you set GLX_X_VISUAL_TYPE_EXT to TrueColor, DirectColor, GrayScale, or StaticGray, visual selection will fail.
  - If GLX_X_VISUAL_TYPE_EXT is not in *attrib_list*, and if all other attributes are equivalent, a PseudoColor visual is chosen in preference to a StaticColor visual.
- If an undefined GLX attribute, or an unacceptable enumerated attribute value is encountered, NULL is returned.

More attributes may be specified in the attribute list. If a visual attribute is not specified, a default value is used. See the **glXChooseVisual()** reference page for more detail.

To free the data returned, use **XFree()**.

Using Transparent Pixels

If you want a visual with transparent pixels, and GLX_RGBA is in *attrib_list*, call **glXChooseVisual()** and specify the GLX_TRANSPARENT_TYPE_EXT value GLX_TRANSPARENT_RGB_EXT.

If you want a transparent visual and GLX_RGBA is not in *attrib_list*, call **glXChooseVisual()** and specify the GLX_TRANSPARENT_TYPE_EXT value GLX_TRANSPARENT_INDEX_EXT.

Don’t specify one of the following values in *attrib_list* because typically only one transparent color or index value is supported:

GLX_TRANSPARENT_INDEX_VALUE_EXT,
GLX_TRANSPARENT_{RED|GREEN|BLUE|ALPHA}_VALUE_EXT

Once you have a transparent visual, you can query the transparent color value by calling **glXGetConfig()**. To get the transparent index value for visuals that support index rendering, use GLX_TRANSPARENT_INDEX_VALUE_EXT. For visuals that support RGBA rendering, use GLX_TRANSPARENT_{RED | GREEN | BLUE}_VALUE_EXT. The visual attribute GLX_TRANSPARENT_ALPHA_VALUE_EXT is included in the extension for future use.

"How to Create Overlays" on page 60 presents an example program that uses a transparent visual for the overlay window.

The Visual Rating Extension

The visual rating extension, EXT_visual_rating, allows servers to export visuals with improved features or image quality, but lower performance or greater system burden, without having to have these visuals selected preferentially. It is intended to ensure that most—but possibly not all—applications get the "right" visual.

You can use this extension during visual selection, keeping in mind that while you will get a good match for most systems, you may not get the best match for all systems.

Using the Visual Rating Extension

To determine the rating for a visual, call **glXGetConfig()** with *attribute* set to GLX_VISUAL_CAVEAT_EXT. **glXGetConfig()** returns the rating of the visual in the parameter *value*: GLX_NONE_EXT or GLX_SLOW_EXT.

If the GLX_VISUAL_CAVEAT_EXT attribute is not specified in the *attrib_list* parameter of **glXChooseVisual()**, preference is given to visuals with no caveats (that is, visuals with the attribute set to GLX_NONE_EXT). If the GLX_VISUAL_CAVEAT_EXT attribute is specified, then **glXChooseVisual()** matches the specified value exactly. For example, if the value is specified as GLX_NONE_EXT, only visuals with no caveats are considered.

The Video Synchronization Extension

The video synchronization extension, SGI_video_sync, allows an application to synchronize drawing with the vertical retrace of a monitor (in the case of an interlaced monitor, the synchronization is actually with the field rate instead). Using the video synchronization extension, an application can put itself to sleep until a counter corresponding to the number of screen refreshes reaches a desired value. The application can also query the current value of the counter.

The system maintains a video sync counter (an unsigned 32-bit integer) for each screen in a system. The counter is incremented upon each vertical retrace.

The counter runs as long as the graphics subsystem is running; it is initialized by the */usr/gfx/gfxinit* command.

Note: A process can query or sleep on the counter only when a direct context is current; otherwise, an error code is returned. See the reference page for more information.

New Functions

glXGetVideoSyncSGI(), glXWaitVideoSyncSGI().

The Swap Control Extension

The swap control extension, SGI_swap_control, allows an application to set a minimum period for buffer swaps (counted in display retrace periods). This is similar to the **swapinterval()** command in IRIS GL.

To set the buffer swap interval, call **glXSwapIntervalSGI()**, specifying the minimum number of retraces between buffer swaps in the *interval* parameter. For example, a value of 2 means that the color buffer is swapped at most every other display retrace.

glXSwapIntervalSGI() affects only buffer swaps for the GLX write drawable for the current context. Note that **glXSwapBuffers()** may be called with a *drawable* parameter that is not the current GLX drawable; in this case **glXSwapIntervalSGI(),** has no effect on that buffer swap.

New Functions

glXSwapIntervalSGI().

The Import Context Extension

The import context extension, EXT_import_context, allows multiple X clients to share an indirect rendering context. The extension also adds some query routines to retrieve information associated with the current context.

To understand this extension, you must first understand direct and indirect rendering. See “Direct and Indirect Rendering” on page 78 for some background information.

Importing a Context

You can use the extension to import another process OpenGL context as follows:

- To retrieve the XID for a GLX context, call **glXGetGLXContextIDEXT()**:

  ```
  GLXContextID glXGetGLXContextIDEXT(const GLXContext ctx)
  ```

  This function is client-side only. No round trip is forced to the server; unlike most X calls that return a value, **glXGetGLXContextIDEXT()** does not flush any pending events.

- To create a GLX context, given the XID of an existing GLX context, call **glXImportGLXContextEXT()**. You can use this function in place of **glXCreateContext()**, to share another process’ indirect rendering context:

  ```
  GLXContext glXImportGLXContextEXT( Display *dpy,
                                     GLXContextID contextID)
  ```

  Only the server-side context information can be shared between X clients; client-side state, such as pixel storage modes, cannot be shared. Thus, **glXImportGLXContextEXT()** must allocate memory to store client-side information.

  This call does not create a new XID. It merely makes an existing XID available to the importing client. The XID goes away when the creating client drops its connection or the ID is explicitly deleted. The object goes away when the XID goes away and the context is not current to any thread.

- To free the client-side part of a GLX context that was created with **glXImportGLXContextEXT()**, call **glXFreeGLXContextEXT()**.

  ```
  void glXFreeGLXContextEXT(Display *dpy, GLXContext ctx)
  ```

  glXFreeGLXContextEXT() does not free the server-side context information or the XID associated with the server-side context.

Retrieving Display and Context Information

Use the extension to retrieve the display of the current context, or other information about the context, as follows:

- To retrieve the current display associated with the current context, call **glxGetCurrentDisplayEXT()**. If there is no current context, NULL is returned. No round trip is forced to the server; unlike most X calls that return a value, **glXGetCurrentDisplayEXT()** does not flush any pending events.
- To obtain the value of a context's attribute, call **glXQueryContextInfoEXT()**. The values and types corresponding to each GLX context attribute are listed in Table 9-1.

Table 9-1 Type and Context Information for GLX Context Attributes

GLX context attribute	Type	Context Information
GLX_SHARE_CONTEXT_EXT	XID	XID of the share list context
GLX_VISUAL_ID_EXT	XID	visual ID
GLX_SCREEN_EXT	int	screen number

New Functions

glXGetCurrentDisplayEXT(), glXGetGLXContextIDEXT(), glXImportGLXContextEXT(), glXFreeGLXContextEXT().

The Framebuffer Configuration Extension

The framebuffer configuration extension, SGIX_fbconfig, provides three new features:

- It introduces a new way to describe the capabilities of a GLX drawable, that is, to describe the resolution of color buffer components and the type and size of ancillary buffers by providing a GLXFBConfig construct.
- It relaxes the "similarity" requirement when associating a current context with a drawable.
- It supports RGBA rendering to one- and two-component windows and GLX pixmaps as well as pbuffers.

Caution: This extension is an SGIX (experimental) extension. The interface may change, or some other details of the extension may change.

Why Use the Framebuffer Configuration Extension?

Use this extension

- if you want to use pbuffers (see "The Pixel Buffer Extension" on page 179)
- if you want to do RGBA rendering to a PseudoColor or GrayScale window
- instead of **glXChooseVisual()**, since it provides visual selection for all GLX drawables, including pbuffers, and incorporates the visual info and visual rating extensions.

This section briefly explores the three new features provided by the extension.

Describing a Drawable With a GLXFBConfig Construct

Currently GLX overloads X visuals so they have additional buffers and other characteristics needed for OpenGL rendering. This extension packages GLX drawables by defining a new construct, a GLXFBConfig, that encapsulates GLX drawable capabilities and has the following properties:

- It may or may not have an associated X visual. If it does have an associated X visual, then it is possible to create windows that have the capabilities described by the GLXFBConfig.

- A particular GLXFBConfig does not need to work with all GLX drawables. For example, it's possible for implementations to export GLXFBConfigs that work only with GLX pixmaps.

Less-Rigid Similarity Requirements When Matching Context and Drawable

In OpenGL 1.0, when you associate a drawable with a GLX context by calling **glXMakeCurrent()**, the two have to be "similar"; that is, they must have been created with the same visual. This extension relaxes the requirement; it only requires the context and drawable to be compatible. This is less restrictive and implies the following:

- The *render_type* attribute for the context must be supported by the GLXFBConfig that the drawable was created with. For example, if the context was created with *render_type* GLX_RGBA_TYPE_SGIX, then the GLX_RENDER_TYPE_SGIX attribute of the GLXFBConfig must have the GLX_RGBA_BIT_SGIX bit set.
- All color buffers and ancillary buffers that exist in both GLXFBConfigs must have the same size. For example, a GLX drawable that has a front left buffer and a back left buffer with red, green, and blue sizes of 4 is not compatible with a GLXFBConfig that has only a front left buffer with red, green, and blue sizes of 8. However, it is compatible with a GLXFBConfig that has only a front left buffer if the red, green, and blue sizes are 4.

Note that when a context is created, it has an associated rendering type: GLX_RGBA_TYPE_SGIX or GLX_COLOR_INDEX_TYPE_SGIX.

Less-Rigid Match of GLX Visual and X Visual

The current GLX specification requires that the GLX_RGBA visual attribute be associated only with TrueColor and DirectColor X visuals. This extension makes it possible to do RGBA rendering to windows created with visuals of type PseudoColor, StaticColor, GrayScale, and StaticGray. In each case, the red component is used to generate the display, and the green and blue components, if present, are ignored for display purposes.

The OpenGL RGBA rendering semantics are more powerful than the OpenGL index rendering semantics. By extending the X visual types that can be associated with an RGBA color buffer, this extension allows RGBA rendering semantics to be used with pseudo-color and gray-scale displays. A particularly useful application of this extension is support of one- and two-component RGBA drawables (a two-component visual has zero-size green and blue components; a one-component visual has, in addition, a zero-size alpha component).

The GLXFBConfig Construct

A GLXFBConfig describes the format, type, and size of the color buffer and ancillary buffers for a GLX drawable. When the GLX drawable is a window, then the GLXFBConfig that describes it has an associated X visual; for a GLXPixmap or GLXPbuffer there may or may not be an X visual associated with the GLXFBConfig.

Choosing a GLXFBConfig Construct

Use **glXChooseFBConfigSGIX()** to get GLXFBConfig constructs that match a list of attributes or to get the list of GLXFBConfig constructs (GLXFBConfigs) that are available on the specified screen.

```
GLXFBConfig *glXChooseFBConfigSGIX(Display *dpy, int screen,
                          const int *attrib_list, int *nitems)
```

If *attrib_list* is NULL, **glXChooseFBConfigSGIX()** returns an array of GLXFBConfigs that are available on the specified screen; otherwise this call returns an array of GLXFBConfigs that match the specified attributes. Table 9-2 shows only attributes specified by this extension; additional attributes are listed on the **glXChooseVisual()** reference page.

Table 9-2 Attributes Introduced by GLXFBConfig

Attribute	Type	Description
GLX_DRAWABLE_TYPE_SGIX	bitmask	Mask indicating which GLX drawables are supported. Valid bits are GLX_WINDOW_BIT_SGIX and GLX_PIXMAP_BIT_SGIX.
GLX_RENDER_TYPE_SGIX	bitmask	Mask indicating which OpenGL rendering modes are supported. Valid bits are GLX_RGBA_BIT_SGIX and GLX_COLOR_INDEX_BIT_SGIX.
GLX_X_RENDERABLE_SGIX	Boolean	True if X can render to drawable.
GLX_FBCONFIG_ID_SGIX	XID	XID of GLXFBConfig.

The attributes are matched in an attribute-specific manner. Some attributes, such as GLX_LEVEL, must match the specified value exactly; others, such as GLX_RED_SIZE, must meet or exceed the specified minimum values. The sorting criteria are defined as follows:

smaller	GLXFBConfigs with an attribute value that meets or exceeds the specified value are matched. Precedence is given to smaller values (when a value is not explicitly requested, the default is implied).
larger	When the value is requested explicitly, only GLXFBConfigs with a corresponding attribute value that meets or exceeds the specified value are matched. Precedence is given to larger values. When the value is not requested, explicitly behaves exactly like the "smaller" criterion.
exact	Only GLXFBConfigs whose corresponding attribute value exactly matches the requested value are considered.
mask	For a config to be considered, all the bits that are set in the requested value must be set in the corresponding attribute. (Additional bits might be set in the attribute.)

Note also that "don't care" means that the default behavior is to have no preference when searching for a matching GLXFBConfig.

Table 9-3 illustrates how each attribute is matched.

Table 9-3 GLXFBConfig Attribute Defaults and Sorting Criteria

Attribute	Default	Sorting Criteria
GLX_BUFFER_SIZE	0	smaller
GLX_LEVEL	0	smaller
GLX_DOUBLEBUFFER	don't care	smaller
GLX_STEREO	False	exact
GLX_AUX_BUFFERS	0	smaller
GLX_RED_SIZE	0	larger
GLX_GREEN_SIZE	0	larger
GLX_BLUE_SIZE	0	larger
GLX_ALPHA_SIZE	0	larger
GLX_DEPTH_SIZE	0	larger
GLX_STENCIL_SIZE	0	larger
GLX_ACCUM_RED_SIZE	0	larger

Table 9-3 GLXFBConfig Attribute Defaults and Sorting Criteria **(continued)**

Attribute	Default	Sorting Criteria
GLX_ACCUM_GREEN_SIZE	0	larger
GLX_ACCUM_BLUE_SIZE	0	larger
GLX_ACCUM_ALPHA_SIZE	0	larger
GLX_SAMPLE_BUFFERS_SGIS	0 if GLX_SAMPLES_ SGIS = 0, 1 otherwise	smaller
GLX_SAMPLES_SGIS	0	smaller
GLX_X_VISUAL_TYPE_EXT	don't care	exact
GLX_TRANSPARENT_TYPE_EXT	GLX_NONE_EXT	exact
GLX_TRANSPARENT_INDEX_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_RED_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_GREEN_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_BLUE_VALUE_EXT	don't care	exact
GLX_TRANSPARENT_ALPHA_VALUE_EXT	don't care	exact
GLX_VISUAL_CAVEAT_EXT	GLX_NONE_EXT	exact, if specified, otherwise minimum.
GLX_DRAWABLE_TYPE_SGIX	GLX_WINDOW_BIT_ SGIX	mask
GLX_RENDER_TYPE_SGIX	GLX_RGBA_BIT_SGIX	mask
GLX_X_RENDERABLE_SGIX	don't care	exact
GLX_FBCONFIG_ID_SGIX	don't care	exact

There are several uses for the **glXChooseFBConfigSGIX()** function:

- Retrieve all GLXFBConfigs on the screen (*attrib_list* is NULL).
- Retrieve a GLXFBConfig with a given ID specified with GLX_FBCONFIG_ID_SGIX.
- Retrieve the GLXFBConfig that's the best match for a given list of visual attributes.

- Retrieve first a list of GLXFBConfigs that match some criteria, for example, each GLXFBConfig available on the screen or all double-buffered visuals available on the screen. Then call **glXGetFBConfigAttribSGIX()** to find their attributes and choose the one that best fits your needs.

Once the GLXFBConfig is obtained, you can use it to create a GLX pixmap, window, or pbuffer (see "The Pixel Buffer Extension" on page 179). In the case of a window, you must first get the associated X visual by calling **glXGetVisualFromFBConfigSGIX().**

Below is a description of what happens when you call **glXChooseFBConfigSGIX()**:

- If no conforming GLXFBConfig exists, or if an error occurs (that is, an undefined GLX attribute is encountered in *attrib_list*, *screen* is invalid, or *dpy* does not support the GLX extension) then NULL is returned.
- If *attrib_list* is not NULL and more than one GLXFBConfig is found, then an ordered list is returned with the GLXFBConfigs that form the "best" match at the beginning of the list. ("How a GLXFBConfig Is Selected" on page 178 describes the selection process.) Use **XFree()** to free the memory returned by **glXChooseFBConfigSGIX()**.
- If GLX_RENDER_TYPE_SGIX is in *attrib_list*, the value that follows is a mask indicating which types of drawables will be created with it. For example, if GLX_RGBA_BIT_SGIX | GLX_COLOR_INDEX_BIT_SGIX is specified as the mask, then **glXChooseFBConfigSGIX()** searches for GLXFBConfigs that can be used to create drawables that work with both RGBA and color index rendering contexts. The default value for GLX_RENDER_TYPE_SGIX is GLX_RGBA_BIT_SGIX.

  The attribute GLX_DRAWABLE_TYPE_SGIX has as its value a mask indicating which drawables to consider. Use it to choose GLXFBConfigs that can be used to create and render to a particular GLXDrawable. For example, if GLX_WINDOW_BIT_SGIX | GLX_PIXMAP_BIT_SGIX is specified as the mask for GLX_DRAWABLE_TYPE_SGIX then **glXChooseFBConfigSGIX()** searches for GLXFBConfigs that support both windows and GLX pixmaps. The default value for GLX_DRAWABLE_TYPE_SGIX is GLX_WINDOW_BIT_SGIX.

If a GLXFBConfig supports windows it has an associated X visual. Use the GLX_X_VISUAL_TYPE_EXT attribute to request a particular type of X visual.

Note that RGBA rendering may be supported for any of the six visual types, but color index rendering can be supported only for PseudoColor, StaticColor, GrayScale, and StaticGray visuals (that is, single-channel visuals). The GLX_X_VISUAL_TYPE_EXT attribute is ignored if GLX_DRAWABLE_TYPE_SGIX is specified in *attrib_list* and the mask that follows does not have GLX_WINDOW_BIT_SGIX set.

GLX_X_RENDERABLE_SGIX is a Boolean indicating whether X can be used to render into a drawable created with the GLXFBConfig. This attribute is always True if the GLXFBConfig supports windows and/or GLX pixmaps.

Retrieving GLXFBConfig Attribute Values

To get the value of a GLX attribute for a GLXFBConfig, call

```
int glXGetFBConfigAttribSGIX(Display *dpy, GLXFBConfig config,
                             int attribute, int *value)
```

If **glXGetFBConfigAttribSGIX()** succeeds, it returns Success, and the value for the specified attribute is returned in *value*; otherwise it returns an error.

Note: A GLXFBConfig has an associated X visual if and only if the GLX_DRAWABLE_TYPE_SGIX value has the GLX_WINDOW_BIT_SGIX bit set.

To retrieve the associated visual, call:

```
XVisualInfo *glXGetVisualFromFBConfigSGIX(Display *dpy,
                                          GLXFBConfig config)
```

If *config* is a valid GLXFBConfig and it has an associated X visual, then information describing that visual is returned; otherwise NULL is returned. Use **XFree()** to free the returned data.

It is also possible to get a GLXFBConfig, given visual information:

```
GLXFBConfig glXGetFBConfigFromVisualSGIX(Display *dpy, XVisualInfo *vis)
```

If the visual is valid and supports OpenGL rendering (that is, if the GLX visual attribute GLX_USE_GL is True) then the associated GLXFBConfig is returned; otherwise NULL is returned.

To create a GLX rendering context or a GLX pixmap using a GLXFBConfig, call **glXCreateContextWithConfigSGIX()** or **glXCreateGLXPixmapWithConfigSGIX()**. The functions are similar to **glXCreateContext()** and **glXCreateGLXPixmap()**. See the reference page for detailed information.

How a GLXFBConfig Is Selected

If more than one GLXFBConfig matches the specification, they are prioritized as follows:

- Preference is given to GLXFBConfigs with the largest GLX_RED_SIZE, GLX_GREEN_SIZE, and GLX_BLUE_SIZE.
- If the requested GLX_ALPHA_SIZE is zero, preference is given to GLXFBConfigs that have GLX_ALPHA_SIZE set to zero; otherwise preference is given to GLXFBConfigs that have the largest GLX_ALPHA_SIZE value.
- If the requested number of GLX_AUX_BUFFERS is zero, preference is given to GLXFBConfigs that have GLX_AUX_BUFFERS set to zero; otherwise preference is given to GLXFBConfigs that have the smallest GLX_AUX_BUFFERS value.
- If the requested size of a particular ancillary buffer is zero (for example, GLX_DEPTH_BUFFER is zero), preference is given to GLXFBConfigs that also have that size set to zero; otherwise preference is given to GLXFBConfigs that have the largest size.
- If the requested value of either GLX_SAMPLE_BUFFERS_SGIS or GLX_SAMPLES_SGIS is zero, preference is given to GLXFBConfigs that also have these attributes set to zero; otherwise preference is given to GLXFBConfigs that have the smallest size.
- If GLX_X_VISUAL_TYPE_EXT is not specified but there is an X visual associated with the GLXFBConfig, the visual type is used to prioritize the GLXFBConfig.
- If GLX_RENDER_TYPE_SGIX has GLX_RGBA_BIT_SGIX set, the visual types are prioritized as follows: TrueColor, DirectColor, PseudoColor, StaticColor, GrayScale, and StaticGray.
- If only the GLX_COLOR_INDEX_SGIX is set in GLX_RENDER_TYPE_SGIX, visual types are prioritized as PseudoColor, StaticColor, GrayScale, and StaticGray.
- If GLX_VISUAL_CAVEAT_EXT is set, the implementation for the particular system on which you run determines which visuals are returned. See “The Visual Rating Extension” on page 167 for more information.

New Functions

glXGetFBConfigAttribSGIX(), glXChooseFBConfigSGIX(), glXCreateGLXPixmapWithConfigSGIX(), glXCreateContextWithConfigSGIX(), glXGetVisualFromFBConfigSGIX(), glXGetFBConfigFromVisualSGIX().

The Pixel Buffer Extension

You can use the pixel buffer extension, SGIX_pbuffer, to define a pixel buffer (GLXPbuffer or pbuffer for short).

Caution: This extension is an SGIX (experimental) extension. The interface or other aspects of the extension may change.

A GLXPbuffer is an additional non-visible rendering buffer for an OpenGL renderer. A pbuffer is equivalent to a GLXPixmap with the following exceptions:

- There is no associated X pixmap. Also, since pbuffers are a GLX resource, it may not be possible to render to them using X or an X extension other than GLX.
- The format of the color buffers and the type and size of associated ancillary buffers for a pbuffer can only be described with a GLXFBConfig; an X visual cannot be used.
- It is possible to create a pbuffer whose contents may be arbitrarily and asynchronously lost at any time.
- A pbuffer works with both direct and indirect rendering contexts.

A pbuffer is allocated in non-visible framebuffer memory, that is, areas for which hardware-accelerated rendering is possible. Applications include additional color buffers for rendering or image processing algorithms.

Pbuffers can be either “volatile,” that is, their contents can be destroyed by another window or pbuffer, or “preserved,” that is, their contents are guaranteed to be correct and are swapped out to virtual memory when other windows need to share the same framebuffer space. The contents of a preserved pbuffer are swapped back in when the pbuffer is needed. The swapping operation incurs a performance penalty, so preserved pbuffers should be used only if re-rendering the contents is not feasible.

A pbuffer is intended to be a “static” resource: a program typically allocates it only once, rather than as a part of its rendering loop. The framebuffer resources that are associated with a GLXPbuffer are also static. They are deallocated only when the GLXPbuffer is destroyed, or, in the case of volatile pbuffers, as the result of X server activity that changes framebuffer requirements of the server.

To create a GLXPbuffer, call **glXCreateGLXPbufferSGIX()**:

```
GLXPbuffer glXCreateGLXPbufferSGIX(Display *dpy, GLXFBConfig config,
                    unsigned int *width, unsigned int *height, int attrib_list)
```

This creates a single GLXPbuffer and returns its XID.

- *width* and *height* specify the pixel width and height of the rectangular GLXPbuffer.
- *attrib_list* specifies a list of attributes for the GLXPbuffer. (Note that the attribute list is defined in the same way as the list for **glXChooseFBConfigSGIX()**: attributes are immediately followed by the corresponding desired value and the list is terminated with None.)

  Currently only two attributes can be specified in *attrib_list*: GLX_CONTENTS_PRESERVED_SGIX and GLX_GET_LARGEST_PBUFFER_SGIX.

  – Use GLX_GET_LARGEST_PBUFFER_SGIX to get the largest available GLXPbuffer when the allocation of the pbuffer would otherwise fail. The width and height of the pbuffer (if one was allocated) are returned in *width* and *height*. Note that these values can never exceed the *width* and *height* that were initially specified. By default, GLX_GET_LARGEST_PBUFFER_SGIX is False.

  – If the GLX_CONTENTS_PRESERVED_SGIX attribute is set to False in *attrib_list*, a "volatile" GLXPbuffer is created and the contents of the pbuffer may be lost at any time. If this attribute is not specified, or if it is specified as True in *attrib_list*, the contents of the pbuffer are preserved, most likely by swapping out portions of the buffer to main memory when a resource conflict occurs. In either case, the client can register to receive a "buffer clobber" event and be notified when the pbuffer contents have been swapped out or have been damaged.

The resulting GLXPbuffer contains color buffers and ancillary buffers as specified by *config*. It is possible to create a pbuffer with back buffers and to swap the front and back buffers by calling **glXSwapBuffers()**. Note that a pbuffer uses framebuffer resources, so applications should deallocate it when not in use.

If **glXCreateGLXPbufferSGIX()** fails to create a GLXPbuffer due to insufficient resources, a BadAlloc X protocol error is generated and NULL is returned. If *config* is not a valid GLXFBConfig then a GLXBadFBConfigSGIX error is generated; if *config* does not support pbuffers, a BadMatch X protocol error is generated.

Rendering to a GLXPbuffer

Any GLX rendering context created with a GLXFBConfig or X visual that is compatible with a GLXFBConfig may be used to render into the pbuffer. For the definition of "compatible," see the description of **glXCreateContextWithConfigSGIX()**, **glXMakeCurrent()** and **glXMakeCurrentReadSGI()**.

If a GLXPbuffer is created with GLX_CONTENTS_PRESERVED_SGIX set to False, the storage for the buffer contents—or a portion of the buffer contents—may be lost at any time. It is not an error to render to a GLXPbuffer that is in this state, but the effect of rendering to it is undefined. It is also not an error to query the pixel contents of such a GLXPbuffer, but the values of the returned pixels are undefined.

Because the contents of a volatile GLXPbuffer can be lost at any time with only asynchronous notification (via the "buffer clobber" event), the only way a client can guarantee that valid pixels are read back with **glReadPixels()** is by grabbing the X server. (Note that this operation is potentially expensive and you should not do it frequently. Also, since this locks out other X clients, you should do it only for short periods of time.) Clients that don't wish to do this can check if the data returned by **glReadPixels()** is valid by calling **XSync()** and then checking the event queue for "buffer clobber" events (assuming that these events were pulled off of the queue before the **glReadPixels()** call).

To destroy a GLXPbuffer call **glXDestroyGLXPbufferSGIX()**:

```
void glXDestroyGLXPbufferSGIX(Display *dpy, GLXPbuffer pbuf)
```

To query an attribute associated with a GLXPbuffer, call **glXQueryGLXPbufferSGIX()**.

```
void glXQueryGLXPbufferSGIX(Display *dpy, GLXPbuffer pbuf, int attribute
                            unsigned int *value)
```

To get the GLXFBConfig for a GLXPbuffer, first retrieve the ID for the GLXFBConfig and then call **glXChooseFBConfigSGIX()**. See "The Framebuffer Configuration Extension" on page 171.

Directing the Buffer Clobber Event

An X client can ask to receive GLX events on a window or GLXPbuffer by calling **glXSelectEventSGIX()**:

```
void glXSelectEventSGIX(Display *dpy, GLXDrawable drawable,
                        unsigned long mask)
```

Currently you can only select the GLX_BUFFER_CLOBBER_BIT_SGIX GLX event as the *mask*. The event structure is

```
typdef struct {
  int event_type;             /* GLX_DAMAGED_SGIX or GLX_SAVED_SGIX */
  int draw_type;              /* GLX_WINDOW_SGIX or GLX_PBUFFER_SGIX */
  unsigned long serial;       /* # of last request processed by server */
  Bool send_event;            /* true if it came for SendEvent request */
  Display *display;           /* display the event was read from */
  GLXDrawable drawable;       /* i.d. of Drawable */
  unsigned int mask;    /* mask indicating which buffers are affected*/
  int x, y;
  int width, height;
  int count;                  /* if nonzero, at least this many more */
} GLXBufferRestoreEvent;
```

A single X server operation can cause several buffer clobber events to be sent, for example, a single GLXPbuffer may be damaged and cause multiple buffer clobber events to be generated. Each event specifies one region of the GLXDrawable that was affected by the X server operation.

A *mask* argument to **glXSelectEventSGIX()** indicates which color and ancillary buffers were affected. All the buffer clobber events generated by a single X server action are guaranteed to be contiguous in the event queue. The conditions under which this event is generated and the event type varies, depend on the type of the GLXDrawable:

- For a preserved GLXPbuffer, a buffer clobber event, with type GLX_SAVED_SGIX, is generated whenever the contents of the GLXPbuffer are swapped out to host memory. The event(s) describes which portions of the GLXPbuffer were affected. Clients who receive many buffer clobber events, referring to different save actions, should consider freeing the GLXPbuffer resource to prevent the system from thrashing due to insufficient resources.
- For a volatile GLXPbuffer, a buffer clobber event with type GLX_DAMAGED_SGIX is generated whenever a portion of the GLXPbuffer becomes invalid. The client may wish to regenerate the invalid portions of the GLXPbuffer.
- For windows, buffer clobber events with type GLX_DAMAGED_SGIX or GLX_SAVED_SGIX occur whenever an ancillary buffer associated with the window is clobbered or swapped out to host memory. The event contains information indicating which color and ancillary buffers—and which portions of those buffers—were affected.

Calling **glXSelectEventSGIX()** overrides any previous event mask that was set by the client for the drawable. Note that it does not affect the event masks that other clients may have specified for a drawable, since each client rendering to a drawable has a separate event mask for it.

To find out which GLX events are selected for a window or GLXPbuffer, call **glXGetSelectedEventSGIX()**:

```
void glXSelectEventSGIX(Display *dpy, GLXDrawable drawable,
                        unsigned long mask)
```

New Functions

glXCreateGLXPbufferSGIX(), glXDestroyGLXPbufferSGIX()
glXGetGLXPbufferStatusSGIX(), glXGetGLXPbufferConfigSGIX(),
glXGetLargestGLXPbufferSGIX().

The Video Source Extension

The video source extension, SGIX_video_source, lets you source pixel data from a video stream to the OpenGL renderer. The video source extension is available only for system configurations that have direct hardware paths from the video hardware to the graphics accelerator. On other systems, you need to transfer video data to host memory and then call **glDrawPixels()** or **glTex{Sub}Image()** to transfer data to the framebuffer or to texture memory.

The video source extension introduces a new type of GLXDrawable—GLXVideoSourceSGIX—that is associated with the drain node of a Video Library (VL) path. A GLXVideoSourceSGIX drawable can only be used as the *read* parameter to **glXMakeCurrentReadSGI()** to indicate that pixel data should be read from the specified video source instead of the framebuffer.

Caution: This extension is an SGIX (experimental) extension. The interface may change, or it may not be supported in future releases.

The remainder of this section presents two examples: Example 9-1 demonstrates the video to graphics capability of the Sirius video board using OpenGL. Example 9-2 is a code fragment for how to use the video source extension to load video into texture memory.

Example 9-1 Use of the Video Source Extension

```
/*
 * vidtogfx.c
 *  This VL program demonstrates the Sirius Video board video->graphics
 *  ability using OpenGL.
 *  The video arrives as fields of an interlaced format.  It is
 *  displayed either by interlacing the previous and the current
 *  field or by pixel-zooming the field in Y by 2.
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vl/vl.h>
#include <vl/dev_sirius.h>
#include <GL/glx.h>
#include "xwindow.h"
#include <X11/keysym.h>

/* Video path variables */
VLServer svr;
VLPath path;
VLNode src;
VLNode drn;
/* Video frame size info */
VLControlValue size;

int F1_is_first;                /* Which field is first */

/* OpenGL/X variables */
Display *dpy;
Window window;
GLXVideoSourceSGIX glxVideoSource;
GLXContext ctx;
GLboolean interlace = GL_FALSE;
/*
 * function prototypes
 */
void usage(char *, int);
void InitGfx(int, char **);
void GrabField(int);
void UpdateTiming(void);
void cleanup(void);
void ProcessVideoEvents(void);
static void loop(void);
```

```
int
main(int argc, char **argv)
{
    int         c, insrc = VL_ANY;
    int         device = VL_ANY;
    short       dev, val;
    /* open connection to VL server */

    if (!(svr = vlOpenVideo(""))) {
        printf("couldn't open connection to VL server\n");
        exit(EXIT_FAILURE);
    }

    /* Get the Video input */
    src = vlGetNode(svr, VL_SRC, VL_VIDEO, insrc);
    /* Get the first Graphics output */
    drn = vlGetNode(svr, VL_DRN, VL_GFX, 0);

    /* Create path   */
    path = vlCreatePath(svr, device, src, drn);
    if (path < 0) {
        vlPerror("vlCreatePath");
        exit(EXIT_FAILURE);
    }
    /* Setup path */
    if (vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE,
                           VL_SHARE) < 0) {
        vlPerror("vlSetupPaths");
        exit(EXIT_FAILURE);
    }
    UpdateTiming();
    if (vlSelectEvents(svr, path,VLStreamPreemptedMask |
                            VLControlChangedMask ) < 0) {
            vlPerror("Select Events");
            exit(EXIT_FAILURE);
    }
    /* Open the GL window for gfx transfers */
    InitGfx(argc, argv);
    /* Begin Transfers */
    vlBeginTransfer(svr, path, 0, NULL);
    /* The following sequence grabs each field and displays it in
     * the GL window.
     */
    loop();
}
```

```
void
loop()
{
  XEvent event;
  KeySym key;
  XComposeStatus compose;
  GLboolean clearNeeded = GL_FALSE;

  while (GL_TRUE) {
    /* Process X events */
    while(XPending(dpy)) {
      XNextEvent(dpy, &event);
      /* Don't really need to handle expose as video is coming at
       * refresh speed.
       */
      if (event.type == case KeyPress) {
        XLookupString(&event.xkey, NULL, 0, &key, NULL);
        switch (key) {
         case XK_Escape:
          exit(EXIT_SUCCESS);
         case XK_i:
          if (hasInterlace) {
            interlace = !interlace;
            if (!interlace) {
              if (!glXMakeCurrentReadSGI(dpy, window,
                                         glxVideoSource, ctx)) {
                fprintf(stderr,
                        "Can't make current to video\n");
                exit(EXIT_FAILURE);
              }
            } else if (!glXMakeCurrent(dpy, window, ctx)) {
              fprintf(stderr,
                      "Can't make window current to context\n");
              exit(EXIT_FAILURE);
            }
            printf("Interlace is %s\n", interlace ? "On" : "Off");
            /* Clear both buffers */
            glClear(GL_COLOR_BUFFER_BIT);
            glXSwapBuffers(dpy, window);
            glClear(GL_COLOR_BUFFER_BIT);
            glXSwapBuffers(dpy, window);
            glRasterPos2f(0, size.xyVal.y - 1);
          } else {
            printf("Graphics interlacing is not supported\n");
          }
```

```
          break;
        }
      }
    }
    ProcessVideoEvents();
    GrabField(0);
    glXSwapBuffers(dpy, window);
    GrabField(1);
    glXSwapBuffers(dpy, window);
  }
}

/*
 * Open an X window of appropriate size and create context.
 */
void
InitGfx(int argc, char **argv)
{
  int i;
  XSizeHints hints;
  int visualAttr[] = {GLX_RGBA, GLX_DOUBLEBUFFER, GLX_RED_SIZE, 12,
                      GLX_GREEN_SIZE, 12, GLX_BLUE_SIZE, 12,
                      None};
  const char *extensions;

  /* Set hints so window size is exactly as the video frame size */
  hints.x = 50; hints.y = 0;
  hints.min_aspect.x = hints.max_aspect.x = size.xyVal.x;
  hints.min_aspect.y = hints.max_aspect.y = size.xyVal.y;
  hints.min_width = size.xyVal.x;
  hints.max_width = size.xyVal.x;
  hints.base_width = hints.width = size.xyVal.x;
  hints.min_height = size.xyVal.y;
  hints.max_height = size.xyVal.y;
  hints.base_height = hints.height = size.xyVal.y;
  hints.flags = USSize | PAspect | USPosition | PMinSize | PMaxSize;
  createWindowAndContext(&dpy, &window, &ctx, 50, 0, size.xyVal.x,
                size.xyVal.y, GL_FALSE, &hints, visualAttr, argv[0]);

  /* Verify that MakeCurrentRead and VideoSource are supported */
  ....
  glxVideoSource = glXCreateGLXVideoSourceSGIX(dpy, 0, svr, path,
                                              VL_GFX, drn);
  if (glxVideoSource == NULL) {
    fprintf(stderr, "Can't create glxVideoSource\n");
```

```
    exit(EXIT_FAILURE);
  }
  if (!glXMakeCurrentReadSGI(dpy, window, glxVideoSource, ctx)) {
    fprintf(stderr, "Can't make current to video\n");
    exit(EXIT_FAILURE);
  }
  /* Set up the viewport according to the video frame size */
  glLoadIdentity();
  glViewport(0, 0, size.xyVal.x, size.xyVal.y);
  glOrtho(0, size.xyVal.x, 0, size.xyVal.y, -1, 1);
  /* Video is top to bottom */
  glPixelZoom(1, -2);
  glRasterPos2f(0, size.xyVal.y - 1);
  glReadBuffer(GL_FRONT);
  /* Check for interlace extension. */
  hasInterlace = ... /* Interlace is supported or not */
}
/*
 * Grab a field. A parameter of  1 = odd Field, 0 = Even Field.
 * Use the global F1_is_first variable to determine how to
 * interleave the fields.
 */
void
GrabField(int odd_field)
{
  /* copy pixels from front to back buffer */
  if (interlace) {
    /* Restore zoom and transfer mode */
    glRasterPos2i(0, 0);
    glPixelZoom(1, 1);
    glCopyPixels(0, 0, size.xyVal.x, size.xyVal.y, GL_COLOR);

    /* Copy the field from Sirius Video to GFX subsystem */
    if (!glXMakeCurrentReadSGI(dpy, window, glxVideoSource, ctx)) {
      fprintf(stderr, "Can't make current to video\n");
      exit(EXIT_FAILURE);
    }
    if (odd_field) {
      if (F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      }
```

```
    } else {
      if (F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      }
    }
#ifdef GL_SGIX_interlace
    if (hasInterlace)
      glEnable(GL_INTERLACE_SGIX);
#endif
    /* video is upside down relative to graphics */
    glPixelZoom(1, -1);
    glCopyPixels(0, 0, size.xyVal.x, size.xyVal.y/2, GL_COLOR);
    if (!glXMakeCurrent(dpy, window, ctx)) {
      fprintf(stderr, "Can't make current to original window\n");
      exit(EXIT_FAILURE);
    }
#ifdef GL_SGIX_interlace
    if (hasInterlace)
      glDisable(GL_INTERLACE_SGIX);
#endif
  } else {
    /* Not deinterlacing */
    glPixelZoom(1, -2);
    if (!odd_field) {
      if (!F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      }
    } else {
      if (!F1_is_first) {
        /* F1 dominant, so odd field is first. */
        glRasterPos2f(0, size.xyVal.y - 2);
      } else {
        /* F2 dominant, so even field is first. */
        glRasterPos2f(0, size.xyVal.y - 1);
      }
    }
```

```
glCopyPixels(0, 0, size.xyVal.x, size.xyVal.y/2, GL_COLOR);
  }
}

/*
 * Get video timing info.
 */
void
UpdateTiming(void)
{
  int is_525;
  VLControlValue timing, dominance;

  /* Get the timing on selected input node */
  if (vlGetControl(svr, path, src, VL_TIMING, &timing) <0) {
    vlPerror("VlGetControl:TIMING");
    exit(EXIT_FAILURE);
  }
  /* Set the GFX Drain to the same timing as input src */
  if (vlSetControl(svr, path, drn, VL_TIMING, &timing) <0) {
    vlPerror("VlSetControl:TIMING");
    exit(EXIT_FAILURE);
  }
  if (vlGetControl(svr, path, drn, VL_SIZE, &size) <0) {
    vlPerror("VlGetControl");
    exit(EXIT_FAILURE);
  }
  /*
   * Read the video source's field dominance control setting and
   * timing, then set a variable to indicate which field has the first
   * line, so that we know how to interleave fields to frames.
   */
  if (vlGetControl(svr, path, src,
                   VL_SIR_FIELD_DOMINANCE, &dominance) < 0) {
    vlPerror("GetControl(VL_SIR_FIELD_DOMINANCE) on video source
                                                         failed");
    exit(EXIT_FAILURE);
  }

  is_525 = ( (timing.intVal == VL_TIMING_525_SQ_PIX) ||
             (timing.intVal == VL_TIMING_525_CCIR601) );

  switch (dominance.intVal) {
    case SIR_F1_IS_DOMINANT:
```

```
      if (is_525) {
        F1_is_first = 0;
      } else {
        F1_is_first = 1;
      }
      break;
    case SIR_F2_IS_DOMINANT:
      if (is_525) {
        F1_is_first = 1;
      } else {
        F1_is_first = 0;
      }
      break;
  }
}

void
cleanup(void)
{
  vlEndTransfer(svr, path);
  vlDestroyPath(svr, path);
  vlCloseVideo(svr);
  exit(EXIT_SUCCESS);
}

void
ProcessVideoEvents(void)
{
  VLEvent ev;

  if (vlCheckEvent(svr, VLControlChangedMask|
                   VLStreamPreemptedMask, &ev) == -1) {
    return;
  }
  switch(ev.reason) {
    case VLStreamPreempted:
      cleanup();
      exit(EXIT_SUCCESS);
    case VLControlChanged:
      switch(ev.vlcontrolchanged.type) {
        case VL_TIMING:
        case VL_SIZE:
        case VL_SIR_FIELD_DOMINANCE:
          UpdateTiming();
          /* change the gl window size */
```

```
            XResizeWindow(dpy, window, size.xyVal.x, size.xyVal.y);
            glXWaitX();
            glLoadIdentity();
            glViewport(0, 0, size.xyVal.x, size.xyVal.y );
            glOrtho(0, size.xyVal.x, 0, size.xyVal.y, -1, 1);
            break;
          default:
            break;
        }
        break;
      default:
        break;
  }
}
```

Example 9-2 Loading Video Into Texture Memory

```
Display *dpy;
Window win;
GLXContext cx;
VLControlValue size, texctl;
int tex_width, tex_height;
VLServer svr;
VLPath path;
VLNode src, drn;

static void init_video_texturing(void)
{
    GLXVideoSourceSGIX videosource;
    GLenum intfmt;
    int scrn;
    float s_scale, t_scale;

    /* set video drain to texture memory */
    drn = vlGetNode(svr, VL_DRN, VL_TEXTURE, 0);

    /* assume svr, src, and path have been initialized as usual */

    /* get the active video area */
    if (vlGetControl(svr, path, src, VL_SIZE, &size) < 0) {
        vlPerror("vlGetControl");
    }
    /* use a texture size that will hold all of the video area */
    /* for simplicity, this handles only 1024x512 or 1024x1024 */
```

```
    tex_width = 1024;
    if (size.xyVal.y > 512) {
        tex_height = 1024;
    } else {
        tex_height = 512;
    }
    /* Set up a texture matrix so that texture coords in 0 to 1    */
    /* range will map to the active video area.  We want           */
    /* s' = s * s_scale                                            */
    /* t' = (1-t) * t_scale  (because video is upside down).       */
    s_scale = size.xyVal.x / (float)tex_width;
    t_scale = size.xyVal.y / (float)tex_height;
    glMatrixMode(GL_TEXTURE);
    glLoadIdentity();
    glScalef(s_scale, -t_scale, 1);
    glTranslatef(0, t_scale, 0);

    /* choose video packing mode */
    texctl.intVal = SIR_TEX_PACK_RGBA_8;
    if (vlSetControl(svr, path, drn, VL_PACKING, &texctl) <0) {
        vlPerror("VlSetControl");
    }
    /* choose internal texture format; must match video packing mode */
    intfmt = GL_RGBA8_EXT;

    glEnable(GL_TEXTURE_2D);
    /* use a non-mipmap minification filter */
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    /* use NULL texture image, so no image has to be sent from host */
    glTexImage2D(GL_TEXTURE_2D, 0, intfmt, tex_width, tex_height, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, NULL);

    if ((videosource = glXCreateGLXVideoSourceSGIX(dpy, scrn, svr,
                                   path, VL_TEXTURE, drn)) == None) {
        fprintf(stderr, "can't create video source\n");
        exit(1);
    }
    glXMakeCurrentReadSGI(dpy, win, videosource, cx);
}

static void draw(void)
{
    /* load video into texture memory */
    glCopyTexSubImage2DEXT(GL_TEXTURE_2D, 0, 0, 0, 0, 0,
                           size.xyVal.x, size.xyVal.y);
```

```
    /* draw the video frame */
    glBegin(GL_POLYGON);
    glTexCoord2f(0,0); glVertex2f(0, 0);
    glTexCoord2f(1,0); glVertex2f(size.xyVal.x, 0);
    glTexCoord2f(1,1); glVertex2f(size.xyVal.x, size.xyVal.y);
    glTexCoord2f(0,1); glVertex2f(0, size.xyVal.y);
    glEnd();

}
```

New Functions

glXCreateGLXVideoSourceSGIX(), glXDestroyGLXVideoSourceSGIX().

Chapter 10

Debugging OpenGL Programs

This chapter explains how to debug graphics applications with the OpenGL debugging tool ogldebug by discussing the following topics:

Ogldebug Overview

The ogldebug tool helps you find OpenGL programming errors and discover OpenGL programming style that may slow down your application. After finding an error, you can correct it and recompile your program.

Using ogldebug, you can perform the following actions at any point during program execution:

- Create a history ("trace") file of all OpenGL calls made. The history file is legal C code and may be compiled and run, letting you experiment with calls to find bugs or try test cases.
- Set a breakpoint at all occurrences of a given OpenGL call.
- Step through (or skip) OpenGL calls.
- Find out about OpenGL errors.
- Display information about OpenGL state, current display lists, and the window that belongs to the application you're debugging.

Note: If you're debugging a multi-window or multi-context application, ogldebug starts a new session (a new window appears) when the application starts a new process. The process ID is displayed in the title bar for each new window.

Note that the OpenGL debugger isn't a general-purpose debugger. Use *dbx* and related tools such as cvd (CASEVision/Workshop Debugger) to find problems in the non-OpenGL portions of a program.

How ogldebug Operates

The OpenGL debugger works like this:

- You invoke ogldebug for an application.
- A special library (*libogldebug.so*) intercepts OpenGL or GLX calls from the application to the OpenGL library (*libGL.so*).
- You can run, halt, step, and trace the application using the ogldebug graphical interface.
- After ogldebug-related processing, the actual OpenGL calls are made as they would have been had ogldebug not been present.

Setting Up ogldebug

Before you can use ogldebug, you must install the *gl_dev.sw.ogldebug* subsystem. You can use the Software Manager from your Toolchest or execute *swmgr* from the command line. Consider also installing *gl_dev.man.ogldebug* to have access to the reference pages.

Using a Configuration File

As you work with ogldebug, you will find that certain settings are best suited for certain situations. You can save and reload groups of ogldebug settings as follows:

- To save settings, choose Save Configuration from the File menu, then enter a filename using the dialog.
- To load settings, choose Load Configuration from the File menu, then select a file using the dialog.

Basic Debugging With ogldebug

This section introduces basic debugging. It first explains how to start ogldebug, then provides step-by-step instructions on creating a trace file to discover OpenGL problems.

Starting ogldebug

To debug your OpenGL program, type on the command line:

`% ogldebug` *program_name*

where *program_name* is the name of your (executable) application.

Note: It's not necessary to compile the application with any special options. The debugger works with any program compiled with *-lGL*.

The debugger brings up its main window (see Figure 10-1) and launches your application; it halts execution just before the first OpenGL call in the application. The window lets you interact with your application and displays information about it.

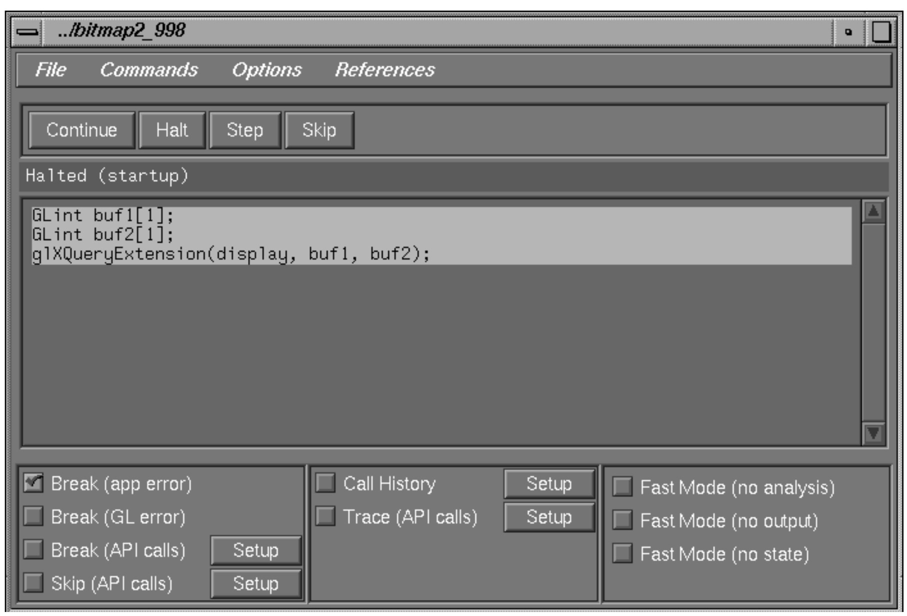


Figure 10-1 The *ogldebug* Window After Startup

Creating a Trace File to Discover OpenGL Problems

A trace file helps you find bugs in the OpenGL portion of your code without having to worry about the mechanics of window operations. Here's an example of how to collect one frame of OpenGL calls:

1. Launch ogldebug:

   `% ogldebug` *your_program_name*

2. Click the Break (API calls) checkbox to select it, then click the Setup button next to it.
3. In the Break Selection panel that appears, select a call that will result in a break after each frame: Either **glXSwapBuffers()** or **glFlush()**, depending on your program.
4. Click the Trace (API calls) checkbox to record the sequence of OpenGL calls.
5. Click the Continue button.

   Execution continues until just before the call(s) selected in the Break Selection panel—see Step 3. The ogldebug tool captures one frame worth of OpenGL calls, embeds them into the X shell window sample program, and saves them under the name *program.*trace.

6. From a shell window, in your source directory, compile the shell:

   `% mv` *program_name*`.trace trace.c`

   `% cc trace.c -lGL -lX11`

7. Launch the new *a.out* program to play back one frame of your application.

You can use *dbx* on the *trace.c* program to track down problems with your OpenGL code without having to deal with the rest of your application.

Note: In the trace file, all GLX calls are commented out with **#ifdef** to prevent them from being compiled unless the ALLOW_GLX_CALLS preprocessor constant is defined. This is because the GLX calls rely on values determined during execution of your application. On the next execution, these values are completely different; they're only provided as reference points, so you can determine where GLX calls were made.

Interacting With ogldebug

Table 10-1 provides more detailed information on working with ogldebug. It explains interacting with ogldebug using buttons or menu commands, interpreting information in display areas, and using checkboxes to determine control flow, tracing, and information collection options (fast mode).

Commands for Basic Interaction

You can perform all basic interaction using the row of buttons just below the menu bar. You can access the same commands using the Commands menu. This section describes each command, including the keyboard shortcut (also listed in the Commands menu).

Table 10-1 Command Buttons and Shortcuts

Command	Result
Continue **Ctrl+C**	Resumes program execution after execution has been stopped (such as after encountering a breakpoint or after you used the Halt or Step command). The program continues running until it reaches another breakpoint or until you explicitly halt it.
Halt **Ctrl+H**	Temporarily stops the application. All state and program information is retained so you can continue execution if you wish.
Step **Ctrl+T**	Continues executing up to the next OpenGL call, then stops before executing that call.
Skip **Ctrl+K**	Skips over the OpenGL command that's about to execute. Useful if you think that command contains an error, or is likely to cause one. The program executes until it reaches the next OpenGL call, then stops.

Understanding Display Areas

Below the row of buttons are two display areas:

- **Status display.** Immediately below the row of buttons is a one-line status display field, where ogldebug posts confirmation of commands and other status indicators.
- **OpenGL display.** Below the status display area is the OpenGL call-information display area. This area shows the next OpenGL command to be executed. Use the scroll bar to look at previously executed commands.

Using Checkboxes

The checkboxes at the bottom of the ogldebug window allow finer control over how information is collected. This section discusses the three groups of available checkboxes:

- Control-Flow Checkboxes
- Trace Checkboxes
- Fast Mode Checkboxes

Control-Flow Checkboxes

Control-flow checkboxes let you determine when a break occurs and which API calls you want to skip. Here's what happens when each of these boxes is checked:

Break (app error)	Your application halts before an OpenGL function is called with an illegal parameter. You must then skip that call to continue.
Break (GL error)	Your application halts because of an OpenGL error.
Break (API calls)	Your application halts before calling any of the OpenGL functions specified using the Setup button. By default, no functions are selected. Figure 10-2 shows a Break Selection Panel with no calls selected so far.
Skip (API calls)	Lets you use the Setup button to specify calls you want to skip. Just as the Break Selection Panel, the skip selection panel lets you create sets of calls for later use.

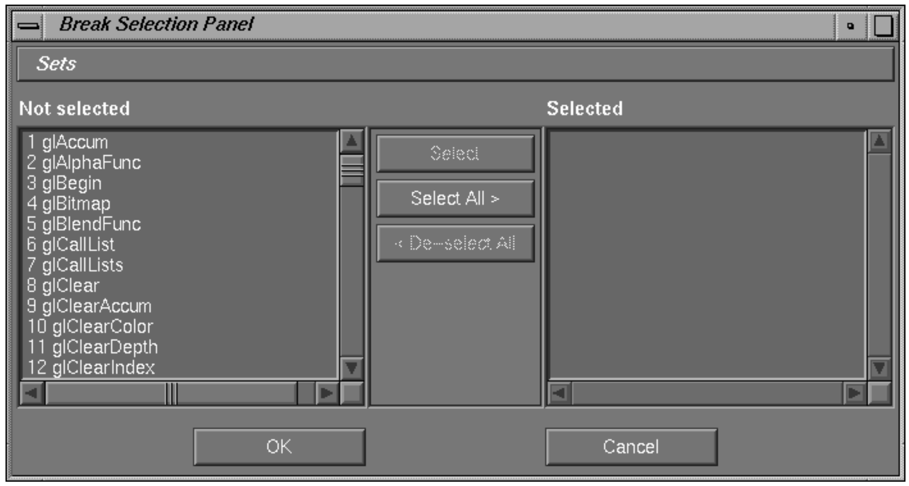


Figure 10-2 Break Selection Panel Menu for Selecting API Calls

Trace Checkboxes

The two Trace checkboxes let you record OpenGL calls in either of two formats:

Trace (API Calls) Collects a trace file in the C-like format displayed in the call information display. For example, choose to select just vertex calls or specifically deselect calls with large output if they are not relevant to the current problem.

Call History Collects the information displayed in the OpenGL display area in a file. The call history also contains some analysis information; during collection, ogldebug looks for frequent program errors or bad programming style and includes that information.

Fast Mode Checkboxes

The Fast Mode checkboxes let you determine which information ogldebug actually collects. If you don't check any of the boxes, execution of your application is quite slow. For example, when state information is being collected, ogldebug makes a **glGet*()** call for every piece of state information after every OpenGL call. Performance degradation is especially noticeable when you're debugging remotely (over a network).

Because of the potential speed penalty, collect only the information you need and turn on fast mode for information you don't need.

Fast Mode (no analysis) Determines whether analysis information is collected.

Fast Mode (no output) Determines whether OpenGL calls are shown in the main display area as they're invoked. Checking this box speeds up program execution but you can no longer follow the program as it's executing.

Fast Mode (no state) Determines whether ogldebug collects OpenGL state information. If it does, you can examine the current state using the State Information command in the Options menu.

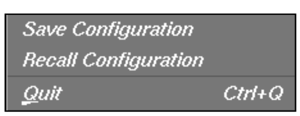
Using Menus to Interact With ogldebug

This section describes how you can interact with ogldebug using menus. You learn about:

- Using the File Menu to Interact With ogldebug
- Using the Commands Menu to Interact With Your Program
- Using the Options Menu to Access Information
- Using the References Menu for Background Information

Using the File Menu to Interact With ogldebug

The File menu lets you save and reload a configuration file, and exit ogldebug, as shown in Figure 10-3.



Saves checkbox settings in a file.
Loads checkbox settings from a file.
Quits ogldebug.

Figure 10-3 The ogldebug File Menu

Using the Commands Menu to Interact With Your Program

The Commands menu (Figure 10-4) provides the same commands as the buttons discussed in “Commands for Basic Interaction” on page 199. You can also find the keyboard accelerators for each command in the menu.

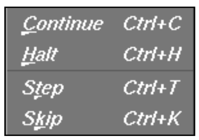


Figure 10-4 The ogldebug Commands Menu

Using the Options Menu to Access Information

The Options menu gives access to some of the information collected by ogldebug (see Figure 10-5).

Here’s a brief description of each command in the Options menu:

Call Count
: Brings up a window with information about the total number of OpenGL calls as well as individual counts for each OpenGL call. You can choose between displaying all available OpenGL calls and all calls that have been made so far.

Display List Information
: Brings up a window with information about the application’s display lists, if any.

Primitive Information
: Provides the current count for all primitives (for example, quads, polygons, and so on).

State Information
: Brings up a window that displays information on OpenGL state variables. To view state information, choose one variable by name. A window displaying the current and default value appears.

Window Information
: Brings up window information for the application you are running from ogldebug.

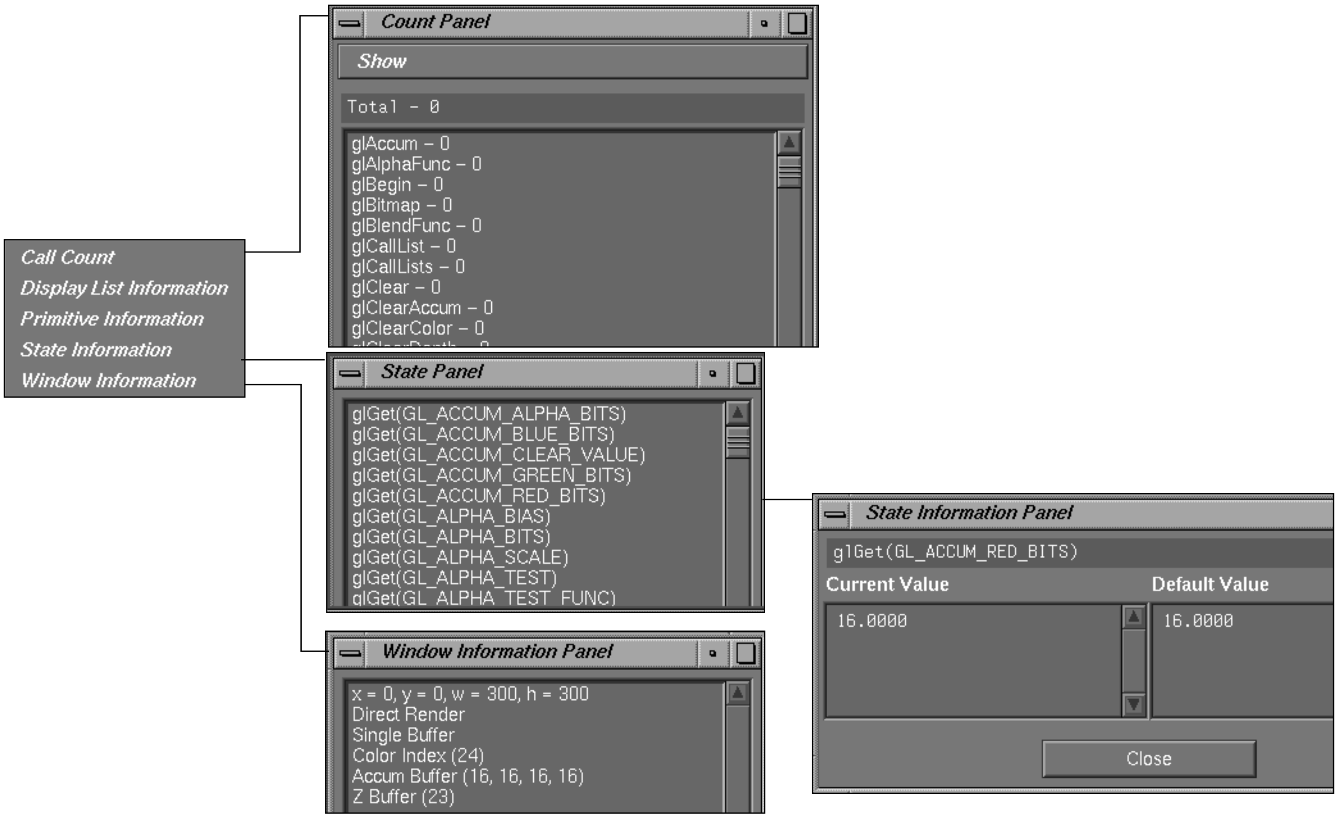


Figure 10-5 The Options Menu and Associated Panels

Using the References Menu for Background Information

The References menu provides access to useful information. You don't use it to interact with ogldebug, but it can facilitate debugging by making needed information available at the click of a mouse. To find a specific item in any list, scroll through the entries (which are arranged in alphabetical order) or use the Search field at the top of the list window.

If you choose Enumerants, a window displays a list of the symbolic names of OpenGL enumerated constants, together with the actual number (in hexadecimal and decimal) that each name represents.

Tips for Debugging Graphics Programs

The following sections are not directly related to the ogldebug debugging tool. They give advice on aspects of debugging that are known to be important for OpenGL debugging. Most points apply primarily to graphics programs and may not be obvious to people used to debugging text-based programs.

General Debugging Tips

Here are some general debugging tips that are useful when debugging an OpenGL program.

- OpenGL never signals errors but simply records them; determining whether an error occurred is up to the user. During the debugging phase, your program should call **glGetError()** to look for errors frequently (for example, once per redraw) until **glGetError()** returns GL_NO_ERROR. While this slows down performance somewhat, it helps you debug the program efficiently. You can use ogldebug to automatically call **glGetError()** after every OpenGL call.
- Use an iterative coding process: add some graphics-related code, build and test to ensure expected results, and repeat as necessary.
- Be careful with OpenGL enumerated constants that are similar in name. For example, **glBegin(GL_LINES)** works; **glBegin(GL_LINE)** does not. Using **glGetError()** can help to detect problems like this (it reports GL_INVALID_ENUM for this specific case).
- Use only per-vertex operations in a **glBegin()/glEnd()** sequence. Within a **glBegin()/glEnd()** sequence, the only graphics commands that may be used are commands for setting materials, colors, normals, edge flags, texture coordinates, surface parametric coordinates, and vertex coordinates. The use of any other graphics command is illegal. The exact list of allowable commands is given in the reference page for **glBegin()**. Even if other calls appear to work, they are not guaranteed to work in the future and may have severe performance penalties.
- If your program is too slow, see the chapters on tuning in this guide for help on increasing performance.
- Check for matching **glPushMatrix()** and **glPopMatrix()** calls.
- Check matrix mode state information. Generally, an application should stay in GL_MODELVIEW mode, but odd visual effects can occur if the matrix mode isn't right.

Specific Problems and Troubleshooting

This section looks at some specific problems frequently encountered by OpenGL users. Note that one generally useful approach is to experiment with an ogldebug trace of the first few frames. See "Creating a Trace File to Discover OpenGL Problems" on page 198 .

Blank Window

A common problem encountered in graphics programming is a blank window. If you find your display doesn't show what you expected, do the following:

- To make sure you're bound to the right window, try clearing the image buffers with **glClear()**. If you can't clear, you may be bound to the wrong window (or no window at all).
- To make sure you're not rendering in the background color, use an unusual color to clear the window with **glClear()**.
- To make sure you're not clipping everything inadvertently, temporarily move the near and far clipping planes to extreme distances (such as 0.001 and 1000000.0). (Note that a range like this is totally inappropriate for actual use in a program).
- Try backing the viewpoint up to see more of the space.
- Check the section "Troubleshooting Transformations" in Chapter 3 of the *OpenGL Programming Guide*.
- Make sure you're using the correct projection matrix.
- Remember that **glOrtho()** and **glPerspective()** calls multiply onto the current projection matrix; they don't replace it.
- In a double-buffered program, make sure you're calling **glXSwapBuffers()**.
- Make sure you requested a depth buffer if the program is using depth buffering.
- Check the aspect ratio of the viewing frustrum. Don't set up your program using:

```
GLfloat aspect = event.xconfigure.width/event.xconfigure.height
                 /* 0 by integer division */
```

Rotation and Translation Problems

- **Z axis direction.** Remember that by default you start out looking down the negative z axis. Unless you move the viewpoint, objects should have negative z coordinates to be visible.

- **Rotation.** Make sure you've translated back to the origin before rotating (unless you intend to rotate about some other point).
- **Transformation order.** First translating, then rotating an object yields a different result than first rotating, then translating. The order of rotation is also important, for example, R(x), R(y), R(z) is not the same as R(z), R(y), R(x).

Depth Buffering Problems

When your program uses depth testing, be sure to:

- Enable depth testing, using **glEnable()** with a GL_DEPTH_TEST argument—depth testing is off by default. Set the depth function to the desired function, using **glDepthFunc()** —the default function is GL_LESS.
- Request a visual that supports a depth buffer. Note that on some platforms a depth buffer is automatically returned for certain color configuration (for example, RGBA on Indy systems), while on other platforms a depth buffer is only returned when one is specifically requested (RealityEngine systems for example). To guarantee that your program is portable, always ask for a depth buffer explicitly.

Animation Problems

- **Double-buffering.** After drawing to the back buffer, make sure you swap buffers with **glXSwapBuffers()**.
- **Observing the image during drawing.** If you have a performance problem and want to see which part of the image takes the longest to draw, use a single-buffered visual. You can then observe the image as it's drawn. If you don't use resources to control visual selection, call **glDrawBuffer()** with a GL_FRONT argument before rendering.

Lighting Problems

- Turn off specular shading in the early debugging stages. It's harder to visualize where specular highlights should be than where diffuse highlights should be.
- For local light sources, try drawing lines from the light source to the object you are trying to light. This helps make sure the spacial and directional nature of the light is right.
- Make sure you have both GL_LIGHTING enabled and the appropriate GL_LIGHT#'s enabled.

- Try enabling GL_NORMALIZE to see whether normals are being scaled and causing lighting problems. This is particularly important if you call **glScale()**.
- Make sure normals are pointing in the right direction.
- Make sure the light is actually at the intended position. Positions are affected by the current model-view matrix. Enabling light without any **glLight(GL_POSITION)** provides a headlight if called before **gluLookAt()** and so on.

X Window System Problems

- X and OpenGL have different notions of the Y direction. X has the origin (0, 0) in the upper left corner of the window; OpenGL has it in the lower left corner. If you try to track the mouse but find that the object is moving in the "wrong" direction vertically, this is probably the cause.
- Display lists defined in one context aren't visible to other contexts unless they explicitly share contexts.
- **glXUseXFont()** creates display lists for characters. The display lists are only visible in contexts that share objects with the one they were created in.

Pixel and Texture Write Problems

- Make sure the pixel storage mode GL_UNPACK_ALIGNMENT is set to the correct value depending on the type of data. For example:

  ```
  GLubyte buf[] = {0x9D, ... 0xA7};
           /* a lot of bitmap images are passed as bytes! */
  glBitmap(w, h, x, y, 0, 0, buf);
  ```

  The default GL_UNPACK_ALIGNMENT is 4. It should be 1 in the case above.

  The same thing applies to textures.

System-Specific Problems

- Make sure you don't exceed implementation-specific resource limits such as maximum projection stack depth.
- When moving an application from a RealityEngine system to a low-end system, make sure you're not using destination alpha planes. The current low-end machines don't support them.

Chapter 11

Tuning Graphics Applications: Fundamentals

Tuning your software makes it use hardware capabilities more effectively. This chapter looks at tuning graphics applications. It discusses pipeline tuning as a conceptual framework for tuning graphics applications, and introduces some other fundamentals of tuning:

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. For more information about the issues you need to consider, and for a tuning example, look at the following chapters in this book:

- Chapter 12, "Tuning the Pipeline"
- Chapter 13, "Tuning Graphics Applications: Examples"
- Chapter 14, "System-Specific Tuning"

After reading these chapters, experiment with the different techniques described to help you decide where to make these trade-offs.

Note: If optimum performance is crucial, consider using the IRIS Performer rendering toolkit. See "Maximizing Performance With IRIS Performer" on page 8.

Why Is Tuning Useful?

Because the hardware of Silicon Graphics systems is so fast, you may think that tuning your application won't make a significant difference in perceived performance. This is often not the case: both the capabilities of your system and the speed at which your application runs are limiting factors. Even the fastest machine can render only as fast as the application can drive it. Simple changes in application code can therefore make a dramatic difference in rendering time. In addition, the flexibility of Silicon Graphics systems provides the opportunity to make the most appropriate tradeoffs between image quality and performance for your application.

What Is Pipeline Tuning?

Traditional software tuning focuses on finding and tuning hot spots, the 10% of the code in which a program spends 90% of its time. Pipeline tuning uses a different approach: it looks for bottlenecks, overloaded stages that are holding up other processes.

At any time, one stage of the pipeline is 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 narrows the bottleneck, or that creates a new bottleneck somewhere else, is the only thing that further degrades performance. The workload can be increased at other parts of the pipeline without degrading performance, as long as that part does not become a new bottleneck. In this way, an application can sometimes be altered to draw a higher-quality image with no performance degradation.

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.

Three-Stage Model of the Graphics Pipeline

The graphics pipeline in all Silicon Graphics workstations consists of three conceptual stages (see Figure 11-1):

1. **The CPU subsystem.** The application program running on the CPU, feeding commands to the graphics subsystem.
2. **The geometry subsystem.** The per-polygon operations, such as coordinate transformations, lighting, texture coordinate generation, and clipping.

3. **The raster subsystem.** The per-pixel operations, such as the simple operation of writing color values into the framebuffer, or more complex operations like depth buffering, alpha blending, and texture mapping.

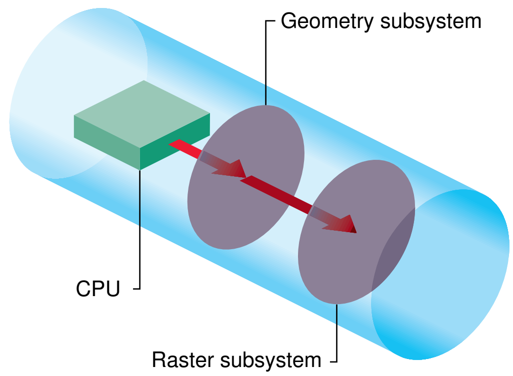


Figure 11-1 Three-Stage Model of the Graphics Pipeline

Note that on some platforms, part of the geometry subsystem is implemented within the CPU subsystem, but the conceptual model is still useful. Note also that 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 geometry operations is lightly loaded. Because those few polygons cover many pixels on the screen, the pipeline stage that does rasterization is heavily loaded.

To speed up this program, you must speed up the rasterization 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 depth-buffering. In addition, 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.

Isolating Bottlenecks in the Pipeline: Overview

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.) If changing the amount of work 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.

- **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, 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 OpenGL calls is a sufficient test. For example, replacing vertex and normal calls like **glVertex3fv()** and **glNormal3fv()** with color subroutine calls like **glColor3fv()** 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. For more information, see "CPU Tuning: Basics" on page 223.

- **Geometry bottlenecks.** Programs that create bottlenecks in the geometry (per-polygon) stage are called *transform limited*. To test for bottlenecks in geometry operations, change the program so that the application code runs at the same speed and the same number of pixels are filled, but the geometry work is reduced. For example, if you are using lighting, call **glDisable()** with a GL_LIGHTING argument to turn off lighting temporarily. If performance improves, your application has a per-polygon bottleneck. For more information, see "Tuning the Geometry Subsystem" on page 238.

- **Rasterization bottlenecks.** Programs that cause bottlenecks at the rasterization (per-pixel) stage in the pipeline are *fill-rate limited*. To test for bottlenecks in rasterization operations, shrink objects or make the window smaller 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 depth-buffering, texturing, or alpha blending. If any of these experiments speeds up the program, it has a per-pixel bottleneck. For more information, see "Tuning the Raster Subsystem" on page 243.

In general, it is easiest to first determine if your application is host (CPU) limited using *osview* and checking whether the CPU usage is near 100%. Then check whether the application is fill (per-pixel) limited by shrinking the window. You then only have to prove that it's geometry limited. The *osview* program also includes statistics that indicate whether the performance bottleneck is in the graphics subsystem or in the host.

Many programs draw a variety of things, each of which stresses different parts of the system. Decompose such a program into pieces and time each piece. You can then focus on tuning the slowest pieces. For an example of such a process, see Chapter 13, "Tuning Graphics Applications: Examples."

Factors Influencing Performance

Pipeline tuning is discussed in detail in Chapter 12, "Tuning the Pipeline." Table 11-1 provides an overview of factors that may limit rendering performance and the part of the pipeline they belong to.

Table 11-1 Factors Influencing Performance

Performance Parameter	Pipeline Stage
Amount of data per polygon	All stages
Time of application overhead	CPU subsystem
Transform rate & mode setting for polygon	Geometry subsystem
Total number of polygons in a frame	Geometry & Raster subsystem
Number of pixels filled	Raster subsystem
Fill rate for the given mode settings	Raster subsystem
Time of screen and/or depth buffer clear	Raster subsystem

Tuning Animation

Tuning animation requires attention to some factors not relevant in other types of applications. This section first looks at factors contributing to animation speed, then provides some advice for optimizing an animation's performance.

Factors Contributing to Animation Speed

The smoothness of an animation depends on its frame rate. The more frames rendered per second, the smoother the motion appears. The basic elements that contribute to the time to render a frame are shown in Table 11-1 above.

Smooth animation also requires double buffering. In double buffering, one framebuffer 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 framebuffer is ready to be displayed, the system swaps the buffers. The system must wait until the next 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.

Frame rates must be integral multiples of the screen refresh time, which is 16.7 msec (milliseconds) 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.

To summarize: 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.

To help monitor timing improvements, turn off double buffering. If you don't, it's difficult to know if you're near a 16.7 msec boundary.

Optimizing Frame Rate Performance

The most important aid for optimizing frame rate performance is taking timing measurements in single-buffer mode only. For more detailed information, see “Taking Timing Measurements” on page 218.

In addition, follow these guidelines to optimize frame rate performance:

- Reduce drawing time to a lower multiple of the screen refresh time. This is the only way to produce an observable performance increase.
- Perform non-graphics computation after **glXSwapBuffers()**.

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

  Clearing a full screen takes time. If you make additional drawing calls immediately after a screen clear, you may fill up the graphics pipeline and force the program to stall. Instead, do some non-drawing work after the clear.

Optimizing Cache and Memory Use

This section first provides some background information about the structure of the cache and about memory lookup. It then gives some tips for optimizing cache and memory use.

How Memory Is Organized

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 referenced, they are automatically copied into higher levels of the hierarchy, so data that is referenced most often migrates to the fastest memory locations.

Here are the areas you should be most concerned about:

- The cache feeds data to the CPU, and cache misses can slow down your program.

  Each processor has instruction caches and 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.
- The translation-lookaside buffer (TLB) keeps track of the location of frequently used pages of memory. If a page translation is not found in the TLB, a delay is incurred while the system looks up the page and enters its translation.

The goal of machine designers and programmers is to maximize the chance of finding data as high up in the memory hierarchy as possible. To achieve this goal, algorithms for maintaining the hierarchy, embodied in the hardware and the operating system, assume that programs have locality of reference in both time and space; that is, programs keep frequently accessed locations close together. Performance increases if you respect 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.

How to Minimize Memory Paging

This section provides some guidelines for minimizing memory paging. You learn about:

- "Minimizing Lookup"
- "Minimizing Cache Misses"
- "Measuring Cache-Miss and Page-Fault Overhead"

Minimizing Lookup

To minimize page lookup, follow these guidelines:

- Keep frequently used data within a minimal number of pages (each page consists of 4K bytes—16K in high-end systems—in current versions of IRIX). Minimize the number of pages referenced in your program by keeping data structures within as few pages as possible. Use *osview* to verify that no TLB misses are occurring.

- Store and access data in flat, sequential data structures, particularly for frequently referenced data. Every pointer indirection could result in the reading of a new page. This is guaranteed to cause performance problems with CPUs like the R10000™ that try to do instructions in parallel.
- In large applications (which cause memory swapping), use **mpin()** to lock important memory into RAM.

Minimizing Cache Misses

Each processor may have first-level instruction and data caches on chip and have second-level cache(s) that are bigger but somewhat slower. The sizes of these caches vary; you can use the *hinv* command to determine the sizes on your system. The first-level data cache is always a subset of the data in the second-level cache.

A cache line is a block of consecutively-addressed words that are treated as a unit by the cache. On a read-miss, a block of several cache lines is read from memory and stored in the cache. The size of this transaction varies from machine to machine (four lines is typical). To minimize cache misses, consider these points:

- Keep frequently accessed data together. This way, the most frequently accessed data remains in the first-level cache wherever possible.
- Access data sequentially. If you're accessing words sequentially, each cache miss brings in sixteen or more 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 simultaneously traversing several large buffers of data, such as an array of vertex coordinates and an array of colors within a loop. There can be cache conflicts between the buffers. Instead, pack the contents into one buffer when possible. If this packing forces a big increase in the size of the data, it may not be the right optimization for that program.

Second-level data cache misses also increase bus traffic, which can be a problem in a multi-processing application. This can happen with multiple processes traversing very large data sets. See "Immediate Mode Drawing Versus Display Lists" on page 224 for additional information.

Measuring Cache-Miss and Page-Fault Overhead

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

- Use *osview* to monitor your application.
- A more rigorous way to estimate the time spent on memory access is to compare the results of PC sampling with those of basic block counting, performing each test with and without calls to **glVertex3fv()**.
  - PC sampling, from *prof*, gives a real-time estimate of the time spent in different sections of the code.
  - Basic block counting, from *prof -pixie*, gives an ideal estimate of how much time should be spent, not including memory references.

  See the *prof* reference page for more information.

  PC sampling includes time for system overhead, so it always predicts longer execution than basic block counting. However, your PC sample time should not be more than 1.5 times the time predicted by *prof -pixie*. For detailed information on profiling, see the *MIPS Compiling and Performance Tuning Guide*, which is available in hardcopy (document number 008-2479-001) or online through IRIS InSight.

  The CASEVision™/WorkShop tools, in particular the performance analyzer, can also help with those measurements. *The WorkShop Overview* introduces the tools.

Taking Timing Measurements

Timing, or benchmarking, parts of your program is an important part of tuning. It helps you determine which changes to your code have a noticeable effect on the speed of your application.

To achieve performance that is demonstrably close to the best the hardware can achieve, you can first follow the more general tuning tips provided in this manual, but you then need to apply a rigorous and systematic analysis. This section looks at some important issues regarding benchmarking:

- "Benchmarking Basics"
- "Achieving Accurate Timing Measurements"
- "Achieving Accurate Benchmarking Results"

Benchmarking Basics

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 are in a strong position both to tune further and to make appropriate decisions considering the speed-versus-quality trade-off.

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 color and depth buffer clear
- time of buffer swap
- time of application overhead
- number of attribute changes and time per change

Achieving Accurate Timing Measurements

Consider these guidelines to get accurate timing measurements:

- Take measurements on a quiet system.

  Verify that no unusual activity is taking place on your system while you take timing measurements. Other graphics programs, background processes, and network activity can distort timing results. For example, do not have *osview*, *gr_osview*, or *Xclock* running while you are benchmarking.

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

  Use a high-resolution clock and make measurements over a period of time that's at least one hundred times the clock resolution. 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's faster, write a loop to execute the test code repeatedly.

Note: Loops like this for timing measurements are highly recommended. Be sure to structure your program in a way that facilitates this approach.

gettimeofday() provides a convenient interface to IRIX clocks with enough resolution to measure graphics performance over several frames. Call **syssgi()** with SGI_QUERY_CYCLECNTR for high-resolution timers. If you can repeat the drawing to make a loop that takes ten seconds or so, a stopwatch works fine and you don't need to alter your program to run the test.

- 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, instead of letting the airplane rotate and taking the benchmark. Once a single frame has been analyzed and tuned, look at frames that stress the graphics pipeline in different ways, then analyze and tune them individually.

- Compare multiple trials.

  Run your program multiple times and try to understand variance in the trials. Variance may be due to other programs running, system activity, prior memory placement, or other factors.

- Call **glFinish()** 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 as soon as the graphics pipeline is ready. The CPU can immediately 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. Call **glFinish()** at the end of your timing trial, just before sampling the clock. Also call **glFinish()** before sampling the clock and starting the trial, to ensure no graphics calls remain in the graphics queue ahead of the process you are timing.

- Consider distortions in double-buffered programs.

  Because buffers can only be swapped during a vertical retrace, there is a period, between the time a **glXSwapBuffers()** call is issued and the next vertical retrace, when a program may not execute any graphics calls. A program that attempts to issue graphics calls during this period is put to sleep until the next vertical retrace. This distorts the accuracy of the timing measurement.

Note: To get accurate numbers, you must perform timing trials in single-buffer mode, with no calls to **glXSwapBuffers()**.

When making timing measurements, use **glFinish()** to ensure that all pixels have been drawn before measuring the elapsed time.

Achieving Accurate Benchmarking Results

To benchmark performance for a particular code fragment, follow these steps:

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 target system for the mode settings you are using.

   Refer to the product literature for the target system to determine some transform and fill rates. Determine others by writing and running small benchmarks.

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: Overview" on page 212.

6. On high-end systems where the processes are pipelined and happen simultaneously, the largest of the three times calculated in steps 3, 4, and 5 determines the overall performance. On low-end systems, you may have to add the time needed for the different processes to arrive at a good estimate.

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.

Ultimately, you need to experiment with different rendering techniques and do repeated benchmarks, especially when the unexpected happens.

Verify some of the suggestions presented in the following chapter. Try some techniques on a small program that you understand and use benchmarks to observe the effects. Figure 11-2 shows how you may actually go through the process of benchmarking and reducing bottlenecks several times. This is also demonstrated by the example presented in Chapter 13, “Tuning Graphics Applications: Examples.”

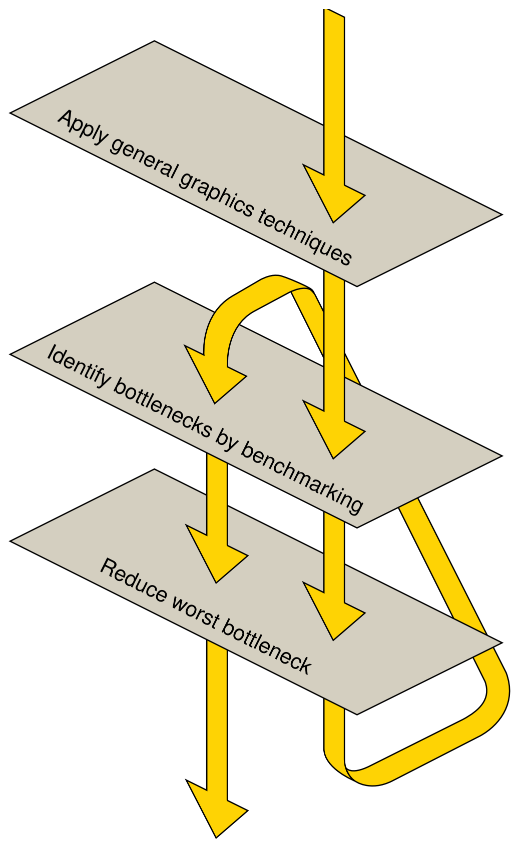


Figure 11-2 Flowchart of the Tuning Process

Chapter 12

Tuning the Pipeline

This chapter looks in some detail at tuning the graphics pipeline. It presents a variety of techniques for optimizing the different parts of the pipeline, providing code fragments and examples as appropriate. You learn about:

CPU Tuning: Basics

The first stage of the rendering pipeline is traversal of the data and sending the current rendering data to the rest of the pipeline. In theory, the entire rendering database (scene graph) must be traversed in some fashion for each frame because both scene content and viewer position can be dynamic.

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

- 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. If you're not compiling with **-xansi** (the default) or **-ansi**, you may need to include **-float** for faster floating-point operations.

  On certain platforms, other compile-time options, such as **-mips3** or **-mips4,** are available.

- 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. Note also that you run a greater risk of cache misses if you flatten the data.

When an application is CPU limited, the entire graphics pipeline may be sitting idle for periods of time. The following sections describe techniques for structuring application code so that the CPU doesn't become the bottleneck.

Immediate Mode Drawing Versus Display Lists

In deciding whether to use immediate mode drawing or display lists, consider the following advantages and disadvantages of display lists. In the end, the characteristics of your application determine which approach to take.

Display lists have the following advantages:

- You don't have to optimize traversal of the data yourself; in fact, display list traversal is typically well-tuned and runs more efficiently than user programs.
- Display lists manage their own data storage. This is particularly useful for algorithmically generated objects.
- Display lists are significantly better for remote graphics over a network. 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. Furthermore, the remote CPU handles much of the responsibility for traversal.
- Display lists are preferable for direct rendering if they contain enough primitives (a total of about ten) because display lists are stored efficiently. If the lists are short, the setup performance cost is not offset by the more efficient storage or saving in CPU time.
- For information on display lists on IMPACT™ systems, see "Using Display Lists Effectively" on page 280.

Display lists do have drawbacks that may affect some applications:

- The most troublesome drawback of display lists is data expansion. To achieve fast, simple traversal on all systems, all data is copied directly into the display list. Therefore, the display list contains an entire copy of all your data plus additional overhead for each command.
- If vertices are shared in structures more complex than the OpenGL primitives (line strip, triangle strip, triangle fan, quad strip), they are stored more than once.
- If the database becomes sufficiently large, paging eventually hinders performance. Therefore, when contemplating the use of OpenGL display lists for really large databases, consider the amount of main memory.
- Compiling display lists may take some time.

For moderate amounts of static data, there is seldom a good reason not to use display lists. Note the reason that the following section on immediate mode drawing is much longer than the section on display lists: Display lists are optimized by the system, and therefore require less work. That alone may be a good reason to prefer display lists under the appropriate circumstances.

CPU Tuning: Immediate Mode Drawing

Immediate mode drawing means that OpenGL commands are executed when they're called, rather than from a display list. This style of drawing 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 optimized subroutines for data traversal. Tuning therefore has two parts:

- "Optimizing the Data Organization"
- "Optimizing Database Rendering Code"

While you may not use each technique in this section, minimize the CPU work done at the per-vertex level and use a simple data structure for rendering traversal.

There is no recipe for writing a peak-performance immediate mode renderer for a specific application. To predict the CPU limitation of your traversal, design potential data structures and traversal loops and write small benchmarks that mimic the memory demands you expect. Experiment with optimizations and benchmark the effects. Experimenting on small examples can save time in the actual implementation.

Optimizing the Data Organization

It is common for scenes to have hierarchical definitions. Scene management techniques may rely on specific hierarchical information. However, a hierarchical organization of the data raises several performance concerns:

- 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; traversing hierarchical structures can therefore access a costly number of pages.

- Complex operations may need access to both the geometric data and other scene information, complicating the data structure.
- Caching behavior is often difficult to predict for dynamic hierarchical data structures.

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.

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

  This is an example of a decision that should be guided by the choice of system on which your application will run. For system-specific tuning information, see Chapter 14, "System-Specific Tuning."

- Balance the data hierarchy. This makes application culling (the process of eliminating objects that don't fall within the viewing frustum) more efficient and effective.

Optimizing Database Rendering Code

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

Ideally, an application spends most of its time traversing the database and sending data to the graphics pipeline. 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 become a bottleneck.

During rendering, 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 have a noticeable effect when the entire frame is rendered.

The rest of this section looks at examples and techniques for optimizing immediate-mode rendering:

- "Examples for Optimizing Data Structures for Drawing"
- "Examples for Optimizing Program Structure"
- "Using Specialized Drawing Subroutines and Macros"
- "Preprocessing Drawing Data: Introduction"
- "Preprocessing Meshes Into Fixed-Length Strips"
- "Preprocessing Vertex Loops"

Examples for Optimizing Data Structures for Drawing

Follow these suggestions for optimizing how your application accesses data:

- **One-Dimensional Arrays**. Use one-dimensional arrays traversed with a pointer that always holds the address for the current drawing command. Avoid array-element addressing or multidimensional array accesses.

  ```
  bad:  glVertex3fv(&data[i][j][k]);
  good: glVertex3fv(dataptr);
  ```

- **Adjacent structures**. 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.
- **Flat structures.** Use flat data structures and do not use multiple pointer indirection when rendering:

```
bad:    glVertex3fv(object->data->vert);
ok:     glVertex3fv(dataptr->vert);
best:   glVertex3fv(dataptr);
```

  The following code fragment is an example of efficient code to draw a single smooth-shaded, 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 **glEnd()** call.

```
glBegin(GL_QUADS);
glNormal3fv(ptr);
glVertex3fv(ptr+4);
glNormal3fv(ptr+8);
glVertex3fv(ptr+12);
glNormal3fv(ptr+16);
glVertex3fv(ptr+20);
glNormal3fv(ptr+24);
glVertex3fv(ptr+28);
glEnd();
ptr += 32;
```

Examples for Optimizing Program Structure

- **Loop unrolling (1).** Avoid short, fixed-length loops, especially around vertices. Instead, unroll these loops:

```
bad:
   for(i=0; i < 4; i++){
        glColor4ubv(poly_colors[i]);
        glVertex3fv(poly_vert_ptr[i]);
   }
good:
   glColor4ubv(poly_colors[0]);
   glVertex3fv(poly_vert_ptr[0]);
   glColor4ubv(poly_colors[1]);
   glVertex3fv(poly_vert_ptr[1]);
   glColor4ubv(poly_colors[2]);
   glVertex3fv(poly_vert_ptr[2]);
   glColor4ubv(poly_colors[3]);
   glVertex3fv(poly_vert_ptr[3]);
```

- **Loop unrolling (2).** Minimize the work done in a loop to maintain and update variables and pointers. Unrolling can often assist in this:

  bad:
  ```
  glNormal3fv(*(ptr++)); glVertex3fv(*(ptr++));
  ```
  or
  ```
  glNormal3fv(ptr); ptr += 4;
  glVertex3fv(ptr); ptr += 4;
  ```
  good:
  ```
  glNormal3fv(*(ptr)); glVertex3fv(*(ptr+1));
  glNormal3fv(*(ptr+2)); glVertex3fv(*(ptr+3));
  ```
  or
  ```
  glNormal3fv(ptr); glVertex3fv(ptr+4); glNormal3fv(ptr+8);
  glVertex3fv(ptr+12);
  ```
  Note: On some processors, such as the R8000,™ loop unrolling may hurt performance more than it helps, so use it with caution. In fact, unrolling too far hurts on any processor because the loop may use an excessive portion of the cache. If it uses a large enough portion of the cache, it may interfere with itself; that is, the whole loop won't fit (not likely) or it may conflict with the instructions of one of the subroutines it calls.

- **Loops accessing buffers.** Minimize the number of different buffers accessed in a loop:

  bad:
  ```
  glNormal3fv(normaldata);
  glTexCoord2fv(texdata);
  glVertex3fv(vertdata);
  ```
  good:
  ```
  glNormal3fv(dataptr);
  glTexCoord2fv(dataptr+4);
  glVertex3fv(dataptr+8);
  ```

- **Loop end conditions.** Make end conditions on loops as trivial as possible; for example, compare the loop variable to a constant, preferably zero. Decrementing loops are often more efficient than their incrementing counterparts:

  bad:
  ```
  for (i = 0; i < (end-beginning)/size; i++)
      {...}
  ```
  better:
  ```
  for (i = beginning; i < end; i += size)
      {...}
  ```

```
good:
    for (i = total; i > 0; i--)
        {...}
```

- **Conditional statements.**
  - Use **switch** statements instead of multiple **if-else-if** control structures.
  - Avoid **if** tests around vertices; use duplicate code instead.
- **Division.** Avoid division. Shift or multiply by a reciprocal instead:

  `f = x * 0.5 instead of f = x / 2.0`

  Integer division is even slower than floating-point division.

  `i = j >> 1 instead of i = j/2`
- **Subroutine prototyping.** Prototype subroutines in ANSI C style to avoid runtime typecasting of parameters:

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

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

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

  Instead, use typecasting of pointers, which occurs at compile time and is efficient:

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

- **Multiple polygons.** Send multiple polygons between **glBegin()/glEnd()** whenever possible:

```
glBegin(GL_TRIANGLES)
....
..../* many triangles */
....
glEnd

glBegin(GL_QUADS)
....
..../* many quads */
....
glEnd
```

Using Specialized Drawing Subroutines and Macros

This section looks at several ways to improve performance by making appropriate choices about display modes, geometry, and so on.

- **Geometry display choices.** Make decisions about which geometry to display and which modes to use at the highest possible level in the program organization.

  The drawing subroutines should be highly specialized leaves in the program's call tree. Decisions made too far down the tree can be redundant. For example, consider a program that switches back and forth between flat-shaded and smooth-shaded drawing. Once this choice has been made for a frame, the decision is fixed and the flag is set. For example, the following code is inefficient:

```
/* Inefficient way to toggle modes */
draw_object(float *data, int npolys, int smooth)  {
int i;
glBegin(GL_QUADS);
for (i = npolys; i > 0; i--) {
    if (smooth) glColor3fv(data);
    glVertex3fv(data + 4);
    if (smooth) glColor3fv(data + 8);
    glVertex3fv(data + 12);
    if (smooth) glColor3fv(data + 16);
    glVertex3fv(data + 20);
    if (smooth) glColor3fv(data + 24);
    glVertex3fv(data + 28);
}
glEnd();
```

  Even though the program chooses the drawing mode before entering the **draw_object()** routine, 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 number of instructions in the disassembled code for a call to **glColor3fv()**, first without, and then with, the **if** test.

  Assembly code for a call without **if** test (six instructions):

```
lw a0,32(sp)
lw t9,glColor3fv
addiu a0,a0,32
jalr ra,t9
nop
lw gp,24(sp)
```

Assembly code for a call with an **if** test (eight instructions):

```
lw t7,40(sp)
beql t7,zero,0x78
nop
lw t9,glColor3fv
lw a0,32(sp)
jalr ra,t9
addiu a0,a0,32
lw gp,24(sp)
```

Notice the two extra instructions required to implement the **if** test. The extra **if** test per vertex increases the number of instructions executed for this otherwise optimal code by 33%. These effects may not be visible if the 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 in the program structure improves performance.

Preprocessing Drawing Data: Introduction

Putting some extra effort into generating a simpler database makes a significant difference when traversing that data for display. A common tendency is to leave the data in a format that is good for loading or generating the object, but not optimal for actually displaying it. For peak performance, 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 when changing from a modeling to a fast-rendering mode. This section discusses "Preprocessing Meshes Into Fixed-Length Strips" and "Preprocessing Vertex Loops" to illustrate this point.

Preprocessing Meshes Into Fixed-Length Strips

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

The following sample code shows a commonly used, but inefficient, way to write a triangle strip render loop:

```
float* dataptr;
...
while (!done) switch(*dataptr) {
    case BEGINSTRIP:
        glBegin(GL_TRIANGLE_STRIP);
        dataptr++;
        break;
    case ENDSTRIP:
        glEnd();
        dataptr++;
        break;
    case EXIT:
        done = 1;
        break;
    default: /* have a vertex !!! */
        glNormal3fv(dataptr);
        glVertex3fv(dataptr + 4);
        dataptr += 8;
}
```

This traversal method 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 triangle strips. Any variation of this code that has per-vertex overhead is likely to be CPU limited for most types of simple graphics operations.

Preprocessing Vertex Loops

Preprocessing is also possible for vertex loops:

```
glBegin(GL_TRIANGLE_STRIP);
for (i=num_verts; i > 0; i--) {
    glNormal3fv(dataptr);
    glVertex3fv(dataptr+4);
    dataptr += 8;
    }
glEnd();
```

For peak immediate mode performance, precompile strips into specialized primitives of fixed length. Only a few fixed lengths are needed. For example, use strips that consist of 12, 8, and 2 primitives.

Note: The optimal length may vary depending on the hardware the program runs on. For more information, see Chapter 14, "System-Specific Tuning."

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

```
/* dump out N 8-triangle strips */
for (i=N; i > 0; i--) {
    glBegin(GL_TRIANGLE_STRIP);
    glNormal3fv(dataptr);
    glVertex3fv(dataptr+4);
    glNormal3fv(dataptr+8);
    glVertex3fv(dataptr+12);
    glNormal3fv(dataptr+16);
    glVertex3fv(dataptr+20);
    glNormal3fv(dataptr+24);
    glVertex3fv(datatpr+28);
    ...
    glEnd();
    dataptr += 64;
}
```

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.

Note: Over-unrolling eventually hurts performance by increasing code size and reducing effectiveness of the instruction cache. The degree of unrolling depends on the processor; run some benchmarks to understand the optimal program structure on your system.

CPU Tuning: Display Lists

In display-list mode, pieces of the database are compiled into static chunks that can then be sent to the graphics pipeline. In this case, the display list is a separate copy of the database that can be stored in main memory in a form optimized for feeding the rest of the pipeline. The database traversal task is to hand the correct chunks to the graphics pipeline. Display lists can be recreated easily with some additional performance cost.

Tuning for display lists focuses mainly on reducing storage requirements. Performance improves if the data fit in the cache because this avoids cache misses when the data is retraversed.

Follow these rules to optimize display lists:

- If possible, compile and execute a display list in two steps instead of using GL_COMPILE_AND_EXECUTE.
- Call **glDeleteLists()** to delete display lists that are no longer needed.

  This frees storage space used by the deleted display lists and expedites the creation of new display lists.
- Avoid duplication of display lists.

  For example, if you have a scene with 100 spheres of different sizes and materials, generate one display list that is a unit sphere centered about the origin. Then for each sphere in the scene, follow these steps:

  1. Set the material for the current sphere.
  2. Issue the necessary scaling and translation commands for sizing and positioning the sphere.
  3. Invoke **glCallList()** to draw the unit sphere display list.

  In this way, a reference to the unit sphere display list is stored instead of all of the sphere vertices for each instance of the sphere.
- Make the display list as flat as possible, but be sure not to exceed the cache size.

  Avoid using an excessive hierarchy with many invocations of **glCallList()**. Each **glCallList()** invocation causes a table lookup to find the designated display list. A flat display list requires less memory and yields simpler and faster traversal. It also improves cache coherency.

  On the other hand, excessive flattening increases the size. For example, if you're drawing a car with four wheels, having a hierarchy with four pointers from the body to one wheel is preferable to a flat structure with one body and four wheels.

Display lists are best used for static objects. 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.

Note: See Chapter 14, "System-Specific Tuning," for potential display list optimizations on the system you are using.

CPU Tuning: Advanced Techniques

After you've applied the techniques discussed in the previous sections, consider using these advanced techniques to tune CPU-limited applications:

- "Mixing Computation With Graphics"
- "Examining Assembly Code"
- "Using Additional Processors for Complex Scene Management"
- "Modeling to the Graphics Pipeline"

Mixing Computation With Graphics

When you are fine-tuning an application, interleaving computation and graphics can make it better balanced and therefore more efficient. Key places for interleaving are after **glXSwapBuffers()**, **glClear()**, and drawing operations that are known to be fill limited (such as drawing a backdrop or a ground plane or any other large polygon).

A **glXSwapBuffers()** call creates a special situation. After calling **glXSwapBuffers()**, an application may be forced to wait for the next vertical retrace (in the worst case, up to 16.7 msecs) before it can issue more graphics calls. For a program drawing 10 frames per second, 15% of the time (worst case) can be spent waiting for the buffer swap 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 after the **glXSwapBuffers()** instead of causing a CPU limitation during drawing.

Clearing the screen is a time-consuming operation. Doing non-graphics computation immediately after the clear is more efficient than sending additional graphics requests down the pipeline and being forced to wait when the pipeline's input queue overflows.

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 a new computation references a large section of data that is not in the data cache, the data for drawing may be swapped out for the computation, then swapped back in for drawing, resulting in worse performance than the original organization.

Examining Assembly Code

When tuning inner rendering loops, examining assembly code can be helpful. Use *dis* 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 an apparently innocuous operation is 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, listed in "Background Reading" on page xxv for additional information.

Using Additional Processors 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.

Using additional processors may also reduce the amount of data rendered for a given frame. Simplifying 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.

Modeling to the Graphics Pipeline

The modeling of the database directly affects the rendering performance of the resulting application and therefore needs to match the performance characteristics of the graphics pipeline and make trade-offs with the database traversals. Graphics pipelines that support connected primitives, such as triangle meshes, benefit from having long meshes in the database. However, the length of the meshes affects the resulting database hierarchy, and long strips through the database do not cull well with simple bounding geometry.

Model objects with an understanding of inherent bottlenecks in the graphics pipeline:

- Pipelines that are severely fill-limited benefit from having objects modeled with cut polygons and more vertices and fewer overlapping parts, which decreases depth complexity.
- Pipelines that are easily geometry- or host-limited benefit from modeling with fewer polygons.

There are several other modeling tricks that can reduce database complexity:

- Use textured polygons to simulate complex geometry. This is especially useful if the graphics subsystem supports the use of textures where the alpha component of the texture marks the transparency of the object. Textures can be used as cut-outs for objects like fences and trees.
- Use textures for simulating particles, such as smoke.
- Use textured polygons as single-polygon billboards. Billboards are polygons that are fixed at a point and rotated about an axis, or about a point, so that the polygon always faces the viewer. Billboards are useful for symmetric objects such as light posts and trees, and also for volume objects such as smoke. Billboards can also be used for distant objects to save geometry. However, the managing of billboard transformations can be expensive and affect both the cull and the draw processes.

Tuning the Geometry Subsystem

This section presents techniques that you can use to tune the geometry subsystem, discussing the following topics:

- "Using Fast Drawing Modes"
- "Using Peak Performance Primitives for Drawing"
- "Optimizing Lighting Performance"
- "Using Expensive Modes Efficiently"
- "Advanced Transform-Limited Tuning Techniques"

Using Fast Drawing Modes

Use flat shading whenever possible. This reduces the number of lighting computations from one per vertex to one per primitive, and also reduces the amount of data that must be passed from the CPU through the graphics pipeline for each primitive. This is particularly important for high-performance line drawing.

Using Peak Performance Primitives for Drawing

This section describes how to draw geometry with optimal primitives. Consider these guidelines to optimize drawing:

- Use connected primitives (line strips, triangle strips, triangle fans, and quad strips). Put at least 8 primitives in a sequence, 12 to 16 if possible.

  Connected primitives are desirable because they reduce the amount of data sent to the graphics subsystem and the amount of per-polygon work done in the pipeline. Typically, about 12 vertices per **glBegin()/glEnd()** are required to achieve peak rates (but this can vary depending on the hardware you're running on). For lines and points, it is especially beneficial to put as many vertices as possible in a **glBegin()/glEnd()** sequence. For information on the most efficient vertex numbers for the system you are using, check Chapter 14, "System-Specific Tuning."

- Use "well-behaved" polygons—convex and planar, with only three or four vertices.

  Concave and self-intersecting polygons must be tessellated by GLU before they can be drawn, and are therefore prohibitively expensive. Nonplanar polygons and polygons with large numbers of vertices are more likely to exhibit shading artifacts.

  If your database has polygons that are not well-behaved, perform an initial one-time pass over the database to transform the troublemakers into well-behaved polygons and use the new database for rendering. Using connected primitives results in additional gains.

- Minimize the data sent per vertex.

  Polygon rates can be affected directly by the number of normals or colors sent per polygon. Setting a color or normal per vertex, regardless of the **glShadeModel()** 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 number of normals and colors per polygon also directly affects the size of a display list containing the object.

- Group like primitives.

Optimizing Lighting Performance

OpenGL offers a large selection of lighting features: Some are virtually “free” in terms of computational time, others offer sophisticated effects with some performance penalty. The penalties some features carry may vary depending on the hardware you’re running on. Be prepared to experiment with the lighting configuration.

As a general rule, use the simplest possible lighting model, a single infinite light with an infinite viewer. For some local effects, try replacing local lights with infinite lights and a local viewer.

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
- RGB mode
- nonlocal viewing—GL_LIGHT_MODEL_LOCAL_VIEWER set to GL_FALSE in **glLightModel()** (the default)
- single-sided lighting—GL_LIGHT_MODEL_TWO_SIDE set to GL_FALSE in **glLightModel()** (the default)
- GL_COLOR_MATERIAL disabled
- GL_NORMALIZE disabled—because this is usually necessary when the model-view matrix includes a scaling transformation, consider preprocessing the scene to eliminate scaling.

Lighting Operations With Noticeable Performance Costs

Follow these guidelines to achieve peak lighting performance:

- Avoid using multiple lights.

  There may be a sharp drop in lighting performance when switching from one light to two lights, but the drop for additional lights is likely to be more gradual.
- Avoid using local lights.

  Local lights are noticeably more expensive than infinite lights.

- Don't change material parameters frequently.

  Changing material parameters can be expensive. If you need to change the material parameters many times per frame, consider rearranging the scene traversal to minimize material changes. Also consider using **glColorMaterial()** to change specific parameters automatically, rather than using **glMaterial()** to change parameters explicitly.

  The following code fragment illustrates how to change ambient and diffuse material parameters at every polygon or at every vertex:

  ```
  glColorMaterial(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);
  glEnable(GL_COLOR_MATERIAL);
  ...
  /* Set ambient and diffuse material parameters: */
  glColor4f(red, green, blue, alpha);
  /* Draw triangles: */
  glBegin(GL_TRIANGLES);
  ...
  glEnd();
  ```

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 more features has still higher cost, the additional penalty is small compared to the initial drop from high-performance lighting. Follow these guidelines to limit use of lighting features that significantly reduce performance:

- Use expensive features with care.

  Setting GL_LIGHT_MODEL_LOCAL_VIEWER to GL_TRUE with **glLightModel()**, while using infinite lights only, reduces performance by a small amount. However, each additional local light noticeably degrades the transform rate.

  Two-sided lighting illuminates both sides of a polygon. This is much faster than the alternative of drawing polygons twice. However, using two-sided lighting is significantly slower than one-sided lighting for a single rendering of an object.

- Use unit-length normals.
- Avoid scaling operations if possible.
- Avoid changing the GL_SHININESS material parameter if possible. Setting a new GL_SHININESS value requires significant computation each time.

Practices to Avoid

Avoid using lighting calls inside a **glBegin()/glEnd()** sequence.

If possible, avoid calls to **glMaterial()** during a **glBegin()/glEnd()** drawing sequence, as this has a serious performance impact. While making such calls to change colors by changing material properties is possible, the performance penalty makes it unadvisable. Use **glColorMaterial()** instead.

Using Expensive Modes Efficiently

OpenGL offers many features that create sophisticated effects with excellent performance. However, these features have some performance cost, compared to drawing the same scene without them. Use these features only where their effects, performance, and quality are justified.

- Turn off features when they are not required.

  Once a feature has been turned on, it can slow the transform rate even when it has no visible effect.

  For example, 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. For these conditions, turn off fog explicitly with

  ```
  glDisable(GL_FOG)
  ```

- Minimize expensive mode changes and sort operations by the most expensive mode. Specifically, consider these tips:
  - Use small numbers of texture maps to avoid the cost of switching between textures. If you have many small textures, consider combining them into a single larger, tiled texture. Rather than switching to a new texture before drawing a textured polygon, choose texture coordinates that select the appropriate small texture tile within the large texture.
  - Avoid changing the projection matrix or changing **glDepthRange()** parameters.
  - When fog is enabled, avoid changing fog parameters.
  - Turn fog off for rendering with a different projection (for example, orthographic) and turn it back on when returning to the normal projection.
- Beware of excessive mode changes, even mode changes considered cheap, such as changes to shade model, depth buffering, and blending function.

Advanced Transform-Limited Tuning Techniques

This section describes advanced techniques for tuning transform-limited drawing. Follow these guidelines to draw objects with complex surface characteristics:

- Use texture to replace complex geometry.

  Textured polygons can be significantly slower than their non-textured counterparts. However, texture can be used instead of extra polygons to add detail to a geometric object. This can greatly simplify geometry, resulting in a net speed increase and an improved picture, as long as it does not cause the program to become fill-limited. Texturing performance varies across the product line, so this technique might not be equally effective on all systems. Experimentation is usually necessary.

- Use **glAlphaFunc()** in conjunction with one or more textures to give the effect of rather complex geometry on a single polygon.

  Consider drawing an image of a complex object by texturing it onto a single polygon. Set alpha values to zero in the texture outside the image of the object. (The edges of the object can be antialiased by using alpha values between zero and one.) Orient the polygon to face the viewer. To prevent pixels with zero alpha values in the textured polygon from being drawn, call:

  ```
  glAlphaFunc(GL_NOTEQUAL, 0.0)
  ```

  This effect is often used to create objects like trees that have complex edges or many holes through which the background should be visible (or both).

- Use culling on a separate processor to eliminate objects or polygons that will be out of sight or too small to see.

Tuning the Raster Subsystem

An explosion of both data and operations is required to rasterize a polygon as individual pixels. Typically, the operations include depth comparison, Gouraud shading, color blending, logical operations, texture mapping, and possibly antialiasing. This section discusses the following techniques for tuning fill-limited drawing:

- "Using Backface/Frontface Removal"
- "Using Expensive Per-Fragment Operations Efficiently"
- "Balancing Polygon Size and Pixel Operations"
- "Clearing the Color and Depth Buffers Simultaneously"

Using Backface/Frontface Removal

To reduce fill-limited drawing, use backface and frontface removal. For example, if you are drawing a sphere, half of its polygons are backfacing at any given time. Backface and frontface removal is done after transformation calculations but before per-fragment operations. This means that backface removal may make transform-limited polygons somewhat slower, but make fill-limited polygons significantly faster. You can turn on backface removal when you are drawing an object with many backfacing polygons, then turn it off again when drawing is completed.

Using Expensive Per-Fragment Operations Efficiently

Use expensive per-fragment operations with care. Per-fragment operations, in order of increasing cost (with flat-shading being the lowest and multisampling the highest) are

1. flat-shading
2. Gouraud shading
3. depth buffering
4. alpha blending
5. texturing
6. multisampling

Each operation can independently slow down the pixel fill rate of a polygon, although depth buffering can help reduce the cost of alpha blending or multisampling for hidden polygons.

Some of this information depends on the particular system the program is running on:

- Texturing is less expensive than alpha blending on new-generation hardware only.
- Alpha blending is less expensive than depth buffering on Indy systems.
- Beware of fill operations that are executed on the host for your graphics platform (for example, texturing on Extreme™ or Elan™ graphics).

Using Depth-Buffering Efficiently

Any rendering operation can become fill-limited 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 depth buffered. It is better to disable depth buffering for the backgrounds and then enable it for other objects where it is needed.

For example, flight simulators use this technique. The sky and ground are drawn with depth buffering disabled, then the polygons lying flat on the ground (runway and grid) are drawn without suffering a performance penalty. Finally, depth buffering is enabled for drawing the mountains and airplanes.

There are many other special cases in which depth buffering might not be required. For example, terrain, ocean waves, and 3D function plots are often represented as height fields (X-Y grids with one height value at each lattice point). It's straightforward to draw height fields in back-to-front order by determining which edge of the field is furthest away from the viewer, then drawing strips of triangles or quadrilaterals parallel to that starting edge and working forward. The entire height field can be drawn without depth testing provided it doesn't intersect any piece of previously-drawn geometry. Depth values need not be written at all, unless subsequently-drawn depth buffered geometry might intersect the height field; in that case, depth values for the height field should be written, but the depth test can be avoided by calling

```
glDepthFunc(GL_ALWAYS)
```

Other Considerations

- Use alpha blending with discretion.

  Alpha blending is an expensive operation. A common use of alpha blending is for transparency, where the alpha value denotes the opacity of the object. For fully opaque objects, disable alpha blending.

- Avoid unnecessary per-fragment operations.

  Turn off per-fragment operations for objects that do not require them, and structure the drawing process to minimize their use without causing excessive toggling of modes.

- Organize drawing to minimize fill operations.

  For example, if a scene has large background polygons, draw them first without depth buffering, then render the more complex depth-buffered objects.

Balancing Polygon Size and Pixel Operations

The pipeline is generally optimized for polygons that are 10 pixels on a side. However, you may need to work with polygons larger or smaller than that depending on the other operations going on in the pipeline:

- If the polygons are too large for the fill-rate to keep up with the rest of the pipeline, the application is fill-rate limited. Smaller polygons balance the pipeline and increase the polygon rate.
- If the polygons are too small for the rest of the pipeline to keep up with filling, then the application is transform limited. Larger and fewer polygons, or fewer vertices, balance the pipeline and increase the fill rate.

Use the simplest possible fill algorithms for drawing very large polygons such as backgrounds.

Clearing the Color and Depth Buffers Simultaneously

The most basic per-frame operations are clearing the color and depth buffers. On some systems, there are optimizations for common, special, cases of these operations.

Whenever you need to clear both the color and depth buffers, don't clear each buffer independently. Instead use

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
```

Tuning the Imaging Pipeline

This section briefly lists some ways in which you can improve pixel processing. Example 13-1 on page 249 provides a code fragment that shows how to set the OpenGL state so that subsequent calls to **glDrawPixels()** or **glCopyPixels()** will be fast.

To improve performance in the imaging pipeline, follow these guidelines:

- Byte-sized components, particularly unsigned byte components, are fast. Use pixel formats where each of the components (red, green, blue, alpha, luminance, or intensity) is 8 bits long.
- Use fewer components, for example, use GL_LUMINANCE_ALPHA or GL_LUMINANCE.

- Use color matrix and color mask to store four luminance values in the RGBA framebuffer. Use color matrix and color mask to work with one component at a time. If one component is being processed, convolution is much more efficient.

  The following code fragment uses the green component as the data source and writes the result of the operation into some (possibly all) of the other components:

```
/* Matrix is in column major order */
GLfloat smearGreenMat[16] = {
    0, 0, 0, 0,
    1, 1, 1, 1,
    0, 0, 0, 0,
    0, 0, 0, 0,
};
/* The variables update R/G/B/A indicate whether the
* corresponding component would be updated.
*/
GLboolean updateR, updateG, updateB, updateA;

...

/* Check for availability of the color matrix extension */

/* Set proper color matrix and mask */
glMatrixMode(GL_COLOR);
glLoadMatrixf(smearGreenMat);
glColorMask(updateR, updateG, updateB, updateA);

/* Perform the imaging operation */
glEnable(GL_SEPARABLE_2D_EXT);
glCopyTexSubImage2DEXT(...);
/* Restore an identity color matrix.  Not needed when the same
* smear operation is to used over and over
*/
glLoadIdentity();

/* Restore previous matrix mode (assuming it is modelview) */
glMatrixMode(GL_MODELVIEW);
...
```

- Load the identity matrix into the color matrix to turn it off.

  When using the color matrix to broadcast one component into all others, avoid manipulating the color matrix with transformation calls such as **glRotate()**. Instead, load the matrix explicitly using **glLoadMatrix().**

- Know where the bottleneck is.

  Similar to polygon drawing, there can be a pixel-drawing bottleneck due to overload in host bandwidth, processing, or rasterizing. When all modes are off, the path is most likely limited by host bandwidth, and a wise choice of host pixel format and type pays off tremendously. This is also why byte components are sometimes faster. For example, use packed pixel format GL_RGB5_A1_EXT to load texture with an GL_RGB5_A1_EXT internal format.

  When either many processing modes or a several expensive modes such as convolution are on, the processing stage is the bottleneck. Such cases benefit from one-component processing, which is much faster than multicomponent processing.

  Zooming up pixels may create a raster bottleneck.

- For simple loading, turn all pixel modes off. Turn depth test and texture off, when possible.
- A big pixel rectangle has a higher throughput (that is, pixels per second) than a small rectangle. Because the imaging pipeline is tuned to trade off a relatively large setup time with a high throughput pixel transfer efficiency, a large rectangle amortizes the setup cost over many pixels, resulting in higher throughput.
- Having no mode changes between pixel operations results in higher throughput. New high-end hardware detects pixel mode changes between pixel operations: when there is no mode change between pixel operations, the setup operation is drastically reduced. This is done to optimize for image tiling where an image is painted on the screen by drawing many small tiles.

Chapter 13

Tuning Graphics Applications: Examples

This chapter first presents a code fragment that helps you draw pixels fast. The second section steps through an example of tuning a small graphics program, showing changes to the program and discussing the speed improvements that result.

Drawing Pixels Fast

The code fragment in Example 13-1 illustrates how to set an OpenGL state so that subsequent calls to **glDrawPixels()** or **glCopyPixels()** will be fast.

Example 13-1 Drawing Pixels Fast

```
/*
 * Disable stuff that's likely to slow down
 * glDrawPixels.(Omit as much of this as possible,
 * when you know in advance that the OpenGL state is
 * already set correctly.)
 */
glDisable(GL_ALPHA_TEST);
glDisable(GL_BLEND);
glDisable(GL_DEPTH_TEST);
glDisable(GL_DITHER);
glDisable(GL_FOG);
glDisable(GL_LIGHTING);
glDisable(GL_LOGIC_OP);
glDisable(GL_STENCIL_TEST);
glDisable(GL_TEXTURE_1D);
glDisable(GL_TEXTURE_2D);
glPixelTransferi(GL_MAP_COLOR, GL_FALSE);
glPixelTransferi(GL_RED_SCALE, 1);
glPixelTransferi(GL_RED_BIAS, 0);
glPixelTransferi(GL_GREEN_SCALE, 1);
glPixelTransferi(GL_GREEN_BIAS, 0);
glPixelTransferi(GL_BLUE_SCALE, 1);
glPixelTransferi(GL_BLUE_BIAS, 0);
glPixelTransferi(GL_ALPHA_SCALE, 1);
```

```
        glPixelTransferi(GL_ALPHA_BIAS, 0);

        /*
         * Disable extensions that could slow down
         * glDrawPixels.(Actually, you should check for the
         * presence of the proper extension before making
         * these calls.I omitted that code for simplicity.)
         */

#ifdef GL_EXT_convolution
        glDisable(GL_CONVOLUTION_1D_EXT);
        glDisable(GL_CONVOLUTION_2D_EXT);
        glDisable(GL_SEPARABLE_2D_EXT);
#endif

#ifdef GL_EXT_histogram
        glDisable(GL_HISTOGRAM_EXT);
        glDisable(GL_MINMAX_EXT);
#endif

#ifdef GL_EXT_texture3D
        glDisable(GL_TEXTURE_3D_EXT);
#endif

        /*
         * The following is needed only when using a
         * multisample-capable visual.
         */

#ifdef GL_SGIS_multisample
        glDisable(GL_MULTISAMPLE_SGIS);
#endif
```

Tuning Example

This section steps you through a complete example of tuning a small program using the techniques discussed in Chapter 12, "Tuning the Pipeline." Consider a program that draws a lighted sphere, shown in Figure 13-1.

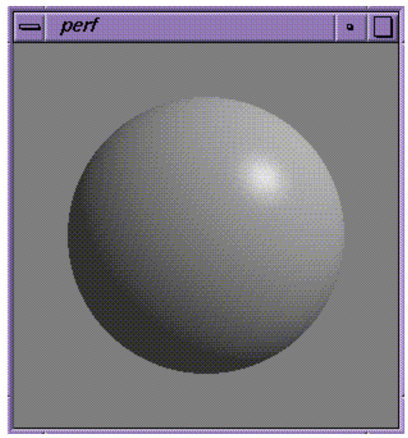


Figure 13-1 Lighted Sphere Created by perf.c

You can use the benchmarking framework in Appendix B, "Benchmarks," for window and timing services. All you need to do is set up the OpenGL rendering context in **RunTest()**, and perform the drawing operations in **Test()**. The first version renders the sphere by drawing strips of quadrilaterals parallel to the sphere's lines of latitude. On a 100MHz Indigo^2 Extreme system, this program renders about 0.77 frames per second.

Example 13-2 Performance Tuning Example Program

```
/*********************************************************************
   cc -o perf -O perf.c -lGLU -lGL -lX11
**********************************************************************/

#include <GL/glx.h>
#include <GL/glu.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <sys/time.h>
#include <math.h>
```

```
char* ApplicationName;
double Overhead = 0.0;
int VisualAttributes[] = { GLX_RGBA, GLX_RED_SIZE, 1, GLX_GREEN_SIZE,
        1, GLX_BLUE_SIZE, 1, GLX_DEPTH_SIZE, 1, None };
int WindowWidth;
int WindowHeight;

/**********************************************************************
 * GetClock - get current time (expressed in seconds)
**********************************************************************/
double
GetClock(void) {
        struct timeval t;

        gettimeofday(&t);
        return (double) t.tv_sec + (double) t.tv_usec * 1E-6;
        }

/**********************************************************************
 * ChooseRunTime - select an appropriate runtime for benchmarking
**********************************************************************/
double
ChooseRunTime(void) {
        double start;
        double finish;
        double runTime;

        start = GetClock();

        /* Wait for next tick: */
        while ((finish = GetClock()) == start)
                ;

        /* Run for 100 ticks, clamped to [0.5 sec, 5.0 sec]: */
        runTime = 100.0 * (finish - start);
        if (runTime < 0.5)
                runTime = 0.5;
        else if (runTime > 5.0)
                runTime = 5.0;

        return runTime;
        }
```

```
/**********************************************************************
 * FinishDrawing - wait for the graphics pipe to go idle
 *
 * This is needed to make sure we're not including time from some
 * previous uncompleted operation in our measurements. (It's not
 * foolproof, since we can't eliminate context switches, but we can
 * assume our caller has taken care of that problem.)
**********************************************************************/
void
FinishDrawing(void) {
        glFinish();
        }


/**********************************************************************
 * WaitForTick - wait for beginning of next system clock tick; return
 * the time
**********************************************************************/
double
WaitForTick(void) {
        double start;
        double current;

        start = GetClock();

        /* Wait for next tick: */
        while ((current = GetClock()) == start)
                ;

        /* Start timing: */
        return current;
        }


/**********************************************************************
 * InitBenchmark - measure benchmarking overhead
 *
 * This should be done once before each risky change in the
 * benchmarking environment. A "risky" change is one that might
 * reasonably be expected to affect benchmarking overhead. (For
 * example, changing from a direct rendering context to an indirect
 * rendering context.)  If all measurements are being made on a single
 * rendering context, one call should suffice.
**********************************************************************/
```

```
void
InitBenchmark(void) {
        double runTime;
        long reps;
        double start;
        double finish;
        double current;

        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /* Wait for the pipe to clear: */
        FinishDrawing();

        /* Measure approximate overhead for finalization and timing
         * routines: */
        reps = 0;
        start = WaitForTick();
        finish = start + runTime;
        do {
                FinishDrawing();
                ++reps;
                } while ((current = GetClock()) < finish);

        /* Save the overhead for use by Benchmark(): */
        Overhead = (current - start) / (double) reps;
        }

/**********************************************************************
 * Benchmark--measure number of caller operations performed per second
 *
 * Assumes InitBenchmark() has been called previously, to initialize
 * the estimate for timing overhead.
 *********************************************************************/
double
Benchmark(void (*operation)(void)) {
        double runTime;
        long reps;
        long newReps;
        long i;
        double start;
        double current;

        if (!operation)
                return 0.0;
```

```
        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /*
         * Measure successively larger batches of operations until we
         * find one that's long enough to meet our runtime target:
         */
        reps = 1;
        for (;;) {
                /* Run a batch: */
                FinishDrawing();
                start = WaitForTick();
                for (i = reps; i > 0; --i)
                        (*operation)();
                FinishDrawing();

                /* If we reached our target, bail out of the loop: */
                current = GetClock();
                if (current >= start + runTime + Overhead)
                        break;

                /*
                 * Otherwise, increase the rep count and try to reach
                 * the target on the next attempt:
                 */
                if (current > start)
                        newReps = reps *(0.5 + runTime /
                                           (current - start - Overhead));
                else
                        newReps = reps * 2;
                if (newReps == reps)
                        reps += 1;
                else
                        reps = newReps;
                }

        /* Subtract overhead and return the final operation rate: */
        return (double) reps / (current - start - Overhead);
        }
/**********************************************************************
 * Test - the operation to be measured
 *
 * Will be run several times in order to generate a reasonably accurate
 * result.
 *********************************************************************/
```

```
void
Test(void) {
        float latitude, longitude;
        float dToR = M_PI / 180.0;

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        for (latitude = -90; latitude < 90; ++latitude) {
                glBegin(GL_QUAD_STRIP);
                for (longitude = 0; longitude <= 360; ++longitude) {
                      GLfloat x, y, z;
                      x = sin(longitude * dToR) * cos(latitude * dToR);
                      y = sin(latitude * dToR);
                      z = cos(longitude * dToR) * cos(latitude * dToR);
                      glNormal3f(x, y, z);
                      glVertex3f(x, y, z);
                      x = sin(longitude * dToR) * cos((latitude+1) *
                                                                dToR);
                      y = sin((latitude+1) * dToR);
                        z = cos(longitude * dToR) * cos((latitude+1) *
                                                                dToR);
                      glNormal3f(x, y, z);
                      glVertex3f(x, y, z);
                      }
                glEnd();
                }
        }

/*********************************************************************
 * RunTest - initialize the rendering context and run the test
*********************************************************************/
void
RunTest(void) {
        static GLfloat diffuse[] = {0.5, 0.5, 0.5, 1.0};
        static GLfloat specular[] = {0.5, 0.5, 0.5, 1.0};
        static GLfloat direction[] = {1.0, 1.0, 1.0, 0.0};
        static GLfloat ambientMat[] = {0.1, 0.1, 0.1, 1.0};
        static GLfloat specularMat[] = {0.5, 0.5, 0.5, 1.0};

        if (Overhead == 0.0)
                InitBenchmark();

        glClearColor(0.5, 0.5, 0.5, 1.0);

        glClearDepth(1.0);
```

```
        glEnable(GL_DEPTH_TEST);

        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
        glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
        glLightfv(GL_LIGHT0, GL_POSITION, direction);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHTING);

        glMaterialfv(GL_FRONT, GL_AMBIENT, ambientMat);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specularMat);
        glMateriali(GL_FRONT, GL_SHININESS, 128);

        glEnable(GL_COLOR_MATERIAL);
        glShadeModel(GL_SMOOTH);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45.0, 1.0, 2.4, 4.6);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        gluLookAt(0,0,3.5,  0,0,0,  0,1,0);

        printf("%.2f frames per second\n", Benchmark(Test));
        }

/**********************************************************************
 * ProcessEvents - handle X11 events directed to our window
 *
 * Run the measurement each time we receive an expose event.
 * Exit when we receive a keypress of the Escape key.
 * Adjust the viewport and projection transformations when the window
 * changes size.
 *********************************************************************/
void
ProcessEvents(Display* dpy) {
        XEvent event;
        Bool redraw = 0;

        do {
                char buf[31];
                KeySym keysym;

                XNextEvent(dpy, &event);
                switch(event.type) {
```

```
                                case Expose:
                                        redraw = 1;
                                        break;
                                case ConfigureNotify:
                                        glViewport(0, 0,
                                                WindowWidth =
                                                      event.xconfigure.width,
                                                WindowHeight =
                                                      event.xconfigure.height);
                                        redraw = 1;
                                        break;
                                case KeyPress:
                                        (void) XLookupString(&event.xkey, buf,
                                                sizeof(buf), &keysym, NULL);
                                        switch (keysym) {
                                                case XK_Escape:
                                                        exit(EXIT_SUCCESS);
                                                default:
                                                        break;
                                                }
                                        break;
                                default:
                                        break;
                                }
                        } while (XPending(dpy));

                if (redraw) RunTest();
                }

/**********************************************************************
 * Error - print an error message, then exit
 *********************************************************************/
void
Error(const char* format, ...) {
        va_list args;

        fprintf(stderr, "%s:  ", ApplicationName);

        va_start(args, format);
        vfprintf(stderr, format, args);
        va_end(args);

        exit(EXIT_FAILURE);
        }
```

```
/**********************************************************************
 * main - create window and context, then pass control to ProcessEvents
 *********************************************************************/
int
main(int argc, char* argv[]) {
        Display *dpy;
        XVisualInfo *vi;
        XSetWindowAttributes swa;
        Window win;
        GLXContext cx;

        ApplicationName = argv[0];

        /* Get a connection: */
        dpy = XOpenDisplay(NULL);
        if (!dpy) Error("can't open display");

        /* Get an appropriate visual: */
        vi = glXChooseVisual(dpy, DefaultScreen(dpy),
                             VisualAttributes);
        if (!vi) Error("no suitable visual");

        /* Create a GLX context: */
        cx = glXCreateContext(dpy, vi, 0, GL_TRUE);

        /* Create a color map: */
        swa.colormap = XCreateColormap(dpy, RootWindow(dpy,
                                   vi->screen), vi->visual, AllocNone);

        /* Create a window: */
        swa.border_pixel = 0;
        swa.event_mask = ExposureMask | StructureNotifyMask |
                                                  KeyPressMask;
        win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0,
                       300, 300, 0, vi->depth, InputOutput, vi->visual,
                       CWBorderPixel|CWColormap|CWEventMask, &swa);
        XStoreName(dpy, win, "perf");
        XMapWindow(dpy, win);

        /* Connect the context to the window: */
        glXMakeCurrent(dpy, win, cx);

        /* Handle events: */
        while (1) ProcessEvents(dpy);
        }
```

Testing for CPU Limitation

An application may be CPU limited, geometry limited, or fill limited. Start tuning by checking for a CPU bottleneck. Replace the **glVertex3f()**, **glNormal3f()**, and **glClear()** calls in **Test()** with **glColor3f()** calls. This minimizes the number of graphics operations while preserving the normal flow of instructions and the normal pattern of accesses to main memory.

```
void
Test(void) {
      float latitude, longitude;
      float dToR = M_PI / 180.0;

      glColor(0, 0, 0);

      for (latitude = -90; latitude < 90; ++latitude) {
            glBegin(GL_QUAD_STRIP);
            for (longitude = 0; longitude <= 360; ++longitude) {
                  GLfloat x, y, z;
                  x = sin(longitude * dToR) * cos(latitude * dToR);
                  y = sin(latitude * dToR);
                  z = cos(longitude * dToR) * cos(latitude * dToR);
                  glColor3f(x, y, z);
                  glColor3f(x, y, z);
                  x = sin(longitude * dToR) * cos((latitude+1) * dToR);
                  y = sin((latitude+1) * dToR);
                  z = cos(longitude * dToR) * cos((latitude+1) * dToR);
                  glColor3f(x, y, z);
                  glColor3f(x, y, z);
                  }
            glEnd();
            }
      }
```

Using the Profiler

The program still renders less than 0.8 frames per second. Because eliminating all graphics output had almost no effect on performance, the program is clearly CPU limited. Use the profiler to determine which function accounts for most of the execution time.

```
% cc -o perf -O -p perf.c -lGLU -lGL -lX11
% perf
% prof perf
-------------------------------------------------------------
Profile listing generated Wed Jul 19 17:17:03 1995
    with:       prof perf
-------------------------------------------------------------

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
    219   2.2s  R4000  R4010 100.0MHz   0     10.0ms     0(bytes)

Each sample covers 4 bytes for every 10.0ms (0.46% of 2.1900sec)
----------------------------------------------------------------------
-p[rocedures] using pc-sampling.
Sorted in descending order by the number of samples in each procedure.
Unexecuted procedures are excluded.
-----------------------------------------------------------------------

samples   time(%)      cum time(%)      procedure (file)

    112   1.1s( 51.1)  1.1s( 51.1)      __sin
                                       (/usr/lib/libm.so:trig.s)
     29  0.29s( 13.2)  1.4s( 64.4)      Test (perf:perf.c)
     18  0.18s(  8.2)  1.6s( 72.6)      __cos (/usr/lib/libm.so:trig.s)
     16  0.16s(  7.3)  1.8s( 79.9)      Finish
                       (/usr/lib/libGLcore.so:../EXPRESS/gr2_context.c)
     15  0.15s(  6.8)  1.9s( 86.8)      __glexpim_Color3f
                       (/usr/lib/libGLcore.so:../EXPRESS/gr2_vapi.c)
     14  0.14s(  6.4)    2s( 93.2)      _BSD_getime
                       (/usr/lib/libc.so.1:BSD_getime.s)
      3  0.03s(  1.4)  2.1s( 94.5)      __glim_Finish
                       (/usr/lib/libGLcore.so:../soft/so_finish.c)
      3  0.03s(  1.4)  2.1s( 95.9)      _gettimeofday
                       (/usr/lib/libc.so.1:gettimeday.c)
      2  0.02s(  0.9)  2.1s( 96.8)      InitBenchmark (perf:perf.c)
      1  0.01s(  0.5)  2.1s( 97.3)      __glMakeIdentity
                       (/usr/lib/libGLcore.so:../soft/so_math.c)
```

```
      1  0.01s(  0.5)  2.1s( 97.7)      _ioctl
                       (/usr/lib/libc.so.1:ioctl.s)
      1  0.01s(  0.5)  2.1s( 98.2)       __glInitAccum64
                       (/usr/lib/libGLcore.so:../soft/so_accumop.c)
      1  0.01s(  0.5)  2.2s( 98.6)       _bzero
                       (/usr/lib/libc.so.1:bzero.s)
      1  0.01s(  0.5)  2.2s( 99.1)       GetClock (perf:perf.c)
      1  0.01s(  0.5)  2.2s( 99.5)       strncpy
                       (/usr/lib/libc.so.1:strncpy.c)
      1  0.01s(  0.5)  2.2s(100.0)      _select
                       (/usr/lib/libc.so.1:select.s)

    219   2.2s(100.0)  2.2s(100.0)        TOTAL
```

Almost 60% of the program's time for a single frame is spent computing trigonometric functions (__sin and __cos).

There are several ways to improve this situation. First consider reducing the resolution of the quad strips that model the sphere. The current representation has over 60,000 quads, which is probably more than is needed for a high-quality image. After that, consider other changes. For example:

- Consider using efficient recurrence relations or table lookup to compute the regular grid of sine and cosine values needed to construct the sphere.
- The current code computes nearly every vertex on the sphere twice (once for each of the two quad strips in which a vertex appears), so you could achieve a 50% reduction in trigonometric operations just by saving and re-using the vertex values for a given line of latitude.

Because exactly the same sphere is rendered in every frame, the time required to compute the sphere vertices and normals is redundant for all but the very first frame. To eliminate the redundancy, generate the sphere just once, and place the resulting vertices and surface normals in a display list. You still pay the cost of generating the sphere once, and eventually may need to use the other techniques mentioned above to reduce that cost, but at least the sphere is rendered more efficiently:

```
void
Test(void) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glCallList(1);
      }
....
```

```
void
RunTest(void){...
      glNewList(1, GL_COMPILE);
      for (latitude = -90; latitude < 90; ++latitude) {
            glBegin(GL_QUAD_STRIP);
            for (longitude = 0; longitude <= 360; ++longitude) {
                  GLfloat x, y, z;
                  x = sin(longitude * dToR) * cos(latitude * dToR);
                  y = sin(latitude * dToR);
                  z = cos(longitude * dToR) * cos(latitude * dToR);
                  glNormal3f(x, y, z);
                  glVertex3f(x, y, z);
                  x = sin(longitude * dToR) * cos((latitude+1) * dToR);
                  y = sin((latitude+1) * dToR);
                  z = cos(longitude * dToR) * cos((latitude+1) * dToR);
                  glNormal3f(x, y, z);
                  glVertex3f(x, y, z);
                  }
            glEnd();
            }
      glEndList();

      printf("%.2f frames per second\n", Benchmark(Test));
      }
```

This version of the program achieves a little less than 2.5 frames per second, a noticeable improvement.

When the **glClear()**, **glNormal3f()**, and **glVertex3f()** calls are again replaced with **glColor3f()**, the program runs at roughly 4 frames per second. This implies that the program is no longer CPU limited, so you need to look further to find the bottleneck.

Testing for Fill Limitation

To check for a fill limitation, reduce the number of pixels that are filled. The easiest way to do that is to shrink the window. If you try that, you see that the frame rate doesn't change for a smaller window, so the program must now be geometry-limited. As a result, it's necessary to find ways to make the processing for each polygon less expensive, or to render fewer polygons.

Working on a Geometry-Limited Program

Previous tests determined that the program is geometry-limited. The next step is therefore to pinpoint the most severe problems and to change the program to alleviate the bottleneck.

Since the purpose of the program is to draw a lighted sphere, you can't eliminate lighting altogether. The program is already using a fairly simple lighting model (a single infinite light and a nonlocal viewer), so there's not much performance to be gained by changing the lighting model.

Smooth Shading Versus Flat Shading

Smooth shading requires more computation than flat shading, so consider changing

```
glShadeModel(GL_SMOOTH);
```

to

```
glShadeModel(GL_FLAT);
```

This increases performance to about 2.75 frames per second. Since this isn't much better than 2.5 frames per second, this discussion continues to use smooth shading.

Reducing the Number of Polygons

Since a change in lighting and shading does not improve performance significantly, the best option is to reduce the number of polygons the program is drawing.

One approach is to tesselate the sphere more efficiently. The simple sphere model used in the program has very large numbers of very small quadrilaterals near the poles, and comparatively large quadrilaterals near the equator. Several superior models exist, but to keep things simple, this discussion continues to use the latitude/longitude tesselation.

A little experimentation shows that reducing the number of quadrilaterals in the sphere causes a dramatic performance increase. When the program places vertices every 10 degrees, instead of every degree, performance skyrockets to nearly 200 frames per second:

```
    for (latitude = -90; latitude < 90; latitude += 10) {
        glBegin(GL_QUAD_STRIP);
        for (longitude = 0; longitude <= 360; longitude += 10) {
```

```
        GLfloat x, y, z;
        x = sin(longitude * dToR) * cos(latitude * dToR);
        y = sin(latitude * dToR);
        z = cos(longitude * dToR) * cos(latitude * dToR);
        glNormal3f(x, y, z);
        glVertex3f(x, y, z);
        x = sin(longitude * dToR) * cos((latitude+10) * dToR);
        y = sin((latitude+10) * dToR);
        z = cos(longitude * dToR) * cos((latitude+10) * dToR);
        glNormal3f(x, y, z);
        glVertex3f(x, y, z);
        }
    glEnd()
    }
```

Of course, this yields a less smooth-looking sphere. When tuning, you often need to make such trade-offs between image quality and drawing performance, or provide controls in your application that allow end users to make the trade-offs.

In this particular case, the improvement up to 200 frames per second becomes apparent only because the program is single-buffered. If the program used double-buffering, performance wouldn't increase beyond the frame rate of the monitor (typically 60 or 72 frames per second), so there would be no performance penalty for using a higher-quality sphere.

If performance is truly critical and sphere intersections aren't likely, consider rendering more vertices at the edge of the silhouette and fewer at the center.

Testing Again for Fill Limitation

If you now shrink the window, performance increases again. This indicates that the program is again fill-limited. To increase performance further, you need to fill fewer pixels, or make pixel-fill less expensive by changing the pixel-drawing mode.

This particular application uses just one special per-fragment drawing mode: depth buffering. Depth buffering can be eliminated in a variety of special cases, including convex objects, backdrops, ground planes, and height fields. Fortunately, since the program is drawing a sphere, you can eliminate depth buffering and still render a correct image by discarding quads that face away from the viewer (the "front" faces, given the orientation of quads in this model):

```
glDisable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

This pushes performance up to nearly 260 frames per second. Further improvements are possible. The program's performance is still far from the upper limit determined by the rate at which the screen can be cleared.

Chapter 14

System-Specific Tuning

This chapter provides tuning information that's relevant for particular Silicon Graphics systems. Use these techniques as needed if you expect your program to be used primarily on one kind of system, or a group of systems. The chapter discusses:

Some points are also discussed in earlier chapters but repeated here because they result in particularly noticeable performance improvement on certain platforms.

Note: To determine your particular hardware configuration, execute */usr/gfx/gfxinfo*. See the reference page for *gfxinfo* for more information. You can also call **glGetString()** with a GL_RENDERER argument. See the reference page for information about the renderer strings for different systems.

Optimizing Performance on Low-End Graphics Systems

This section discusses how you can get the best results from your application on low-end graphics systems, such as the Indy™, Indigo™ Entry, and Indigo^2™ XL systems (but not other Indigo^2 systems); discussing the following topics:

- "Choosing Features for Optimum Performance"
- "Using the Pipeline Effectively"
- "Using Geometry Operations Effectively"
- "Using Per-Fragment Operations Effectively"

Choosing Features for Optimum Performance

By emphasizing features implemented in hardware, you can significantly influence the performance of your application. As a rule of thumb, consider the following:

- **Hardware-supported features**: Lines, filled rectangles, color shading, alpha blending, alpha function, antialiased lines (color-indexed and RGB), line and polygon patterns, color plane masks, color dithering, logical operations selected with **glLogicOp()**, pixel reads and writes, screen to screen copy, and scissoring.
- **Software-supported features**: All features not in hardware, such as stencil and accumulation buffer, fogging and depth queuing, transforms, lighting, clipping, depth buffering, and texturing. Triangles and polygons are partially software supported.

Using the Pipeline Effectively

The low-end graphics systems' FIFO allows the CPU and the graphics subsystem to work in parallel. For optimum performance, follow these guidelines:

- Make sure the graphics subsystem always has enough in the queue.
- Let the CPU perform preprocessing or non-graphic aspects of the application while the graphics hardware works on the commands in the FIFO.

  For example, a full screen clear takes about 3 ms. Let the application do something else immediately after a clear, the FIFO otherwise fills up and forces a stall.

Note that FIFOs in low-end systems are much smaller than those in high-end systems. Not all graphics processing happens in hardware, and the time spent therefore differs greatly. To detect imbalances between the CPU and the graphics FIFO, execute the *gr_osview* command and observe gfxf in the CPU bar and fiwt and finowt in the gfx bar.

- gfxf: time spent waiting for the graphics FIFO to drain.
- fiwt: FIFO filled up and host went to sleep waiting for it to drain.
- finowt: FIFO filled up but drained fast enough that host continued.

Using Geometry Operations Effectively

If your application seems transform limited, you can improve it by considering the tips in this section. The section starts with some general points, then discusses optimizing line drawing and using triangles and polygons effectively.

To improve performance in the geometry subsystem, follow these guidelines:

- Use single-precision floating point parameters for vertices, normals, and colors.
- Transform paths use single-precision floats—it's fastest to use **glVertex3fv()** and **glVertex2fv()**.
- Use **glOrtho()** and a modelview matrix without rotation for best performance.
- Perspective transforms that require multiplication by 1/W or division by W are much slower.
- To minimize the time used for floating point conversion, use floating point coordinates where possible, except where memory size is critical.
- Don't enable normalizing of normals if the modelview matrix doesn't include scaling and if you have unit-length normals.

Optimizing Line-Drawing

Even on low-end systems, lines can provide real-time interactivity. Consider these guidelines:

- Use line drawing while the scene is changing and solid rendering when the scene becomes static.
- Shaded lines and antialiased lines that are one pixel wide are supported by the hardware. Patterned lines are as fast as solid lines.
- Wide lines are drawn as multiple parallel offset lines.
- Depth-queued lines are about as fast as shaded lines.
- The hardware can usually draw lines faster than the software can produce commands, though long or antialiased lines can cause a backup in the graphics pipeline.
- Avoid depth buffering for lines; incorrect depth-sorting artifacts are usually not noticeable.

Optimizing Triangles and Polygons

When rendering triangles and polygons, keep in mind the following:

- Maximize the number of vertices between **glBegin()** and **glEnd().**
- Decompose quads and polygons into triangle strips. The GL_TRIANGLE_STRIP primitive has the fastest path.
- Use connected primitives (triangle, quad, and line strips). Use triangle strips wherever possible and draw as many triangles as possible per **glBegin()/glEnd()** pair.
- When rendering solid triangles, consider the following:
  - Color shading and alpha blending are performed in hardware on Indy and Indigo^2 XL systems. Consult system-specific documentation for information on other low-end systems.
  - Larger triangles have a better overall fill rate than smaller ones because CPU setup per triangle is independent of triangle size.

Using Per-Fragment Operations Effectively

This section looks at some things you can do if your application is fill limited. It provides information about getting the optimum fill rates and about using pixel operations effectively.

Getting the Optimum Fill Rates

To achieve the fastest fill rates possible, consider the following:

- Clear operations and screen-aligned **glRect()** calls that don't use the depth or stencil buffer have a maximum fill rate of 400 MB/sec

  The hardware accelerates drawing rectangles that have their vertical and horizontal edges parallel to those of the window. The OpenGL **glRect()** call—as opposed to IRIS GL **rect()**—specifies rectangles, but depending on the matrix transformations they may not be screen-aligned. If the matrices are such that the rectangle drawn by **glRect()** is screen-aligned, OpenGL detects that and uses the accelerated mode for screen-aligned rectangles.
- Use **glShadeModel()** with GL_FLAT whenever possible, especially for lines.

- Using dithering, shading, patterns, logical operations, writemasks, stencil buffering, and depth buffering (and alpha blending on some systems) slows down an application.
- Use **glEnable()** with GL_CULL_FACE to eliminate backfacing polygons, especially in modes that have slow fill rates, such as depth buffering and texturing (alpha-blending on some systems).
- In any OpenGL matrix mode, low-end systems check for transforms that only scale, and have no rotations or perspective. The system looks at the specified matrices, and if they only scale and have no rotation or perspective, performs optimizations that speed up transformation of vertices to device coordinates. One way to specify this is:

  ```
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0,xsize,0,ysize);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glShadeModel(GL_FLAT);
  ```

  You also have to use a **glVertex2fv()** call to specify 2D vertices.
- Starting with IRIX 6.2, texture mapping speed is increased by about 10 times (compared to previous releases) when texture parameters are specified as follows:

  ```
  glEnable(GL_TEXTURE_2D);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST);
  ```

  In addition, follow these guidelines:

  – For RGB textures, make sure the texture environment mode, set with **glTexEnv(),** is either GL_REPLACE_EXT or GL_DECAL.

  – For RGBA textures, make sure the texture environment mode is GL_REPLACE_EXT.

    Note that the above fast path does not work when stencil is enabled and when depth buffering and alpha testing are both on.

Note: Avoid using depth buffering whenever possible, because the fill rates for depth buffering are much slower. In particular, avoid using depth-buffered lines, as they are especially slow. The depth buffer is as large as the window, so use the smallest possible window to minimize the amount of memory allocated to the depth buffer. The same applies for stencil buffers.

Using Per-Fragment Operations Effectively

Write your OpenGL program to use the combinations of pixel formats and types listed in Table 14-1, for which the hardware can use DMA. The CPU has to reformat pixels in other format and type combinations.

Table 14-1 Pixel Formats and Types Using DMA on Low-End Systems

Format	Type
GL_RGBA	GL_UNSIGNED_BYTE
GL_ABGR_EXT	GL_UNSIGNED_BYTE
GL_COLOR_INDEX	GL_UNSIGNED_BYTE
GL_COLOR_INDEX	GL_UNSIGNED_SHORT

Note that GL_ABGR_EXT provides better performance than GL_RGBA on Indigo Entry systems but not on Indy or Indigo^2 XL where the two perform about the same.

Here are some additional guidelines for optimizing pixel operations:

- **Scrolling.** When scrolling scenes, use **glCopyPixels()** to copy from one scene to the next.

  When you scroll something, such as a teleprompter text scroll or an EKG display, use **glCopyPixels()** to shift the part of the window in the scrolling direction, and draw only the area that's newly exposed.

  This is much faster than completely redrawing each frame.
- **Minimizing calls.** Make each pixel operation draw as much data as possible. For each call, a certain amount of setup is required; you cut down on that time if you minimize the number of calls.
- **Zooming.** Zoomed pixels can't use DMA. A 2 x 2 zoom is faster than other zoom operations.

- **Depth and scissoring.** Low-end systems use an accelerated algorithm that makes clearing the depth buffer virtually free. However, this has slowed enabling and disabling scissoring and changing the scissor rectangle. The larger the scissor rectangle, the longer the delay. Note that rendering while scissoring is turned on is fast; calling **glEnable()** and **glDisable()** with GL_SCISSOR and calling **glScissor()** is slow. This also includes pushing and popping attributes that may cause a scissor change.

Low-End Specific Extensions

For Indy and Indigo^2 XL systems, an extension has been developed that increases fill rate by drawing pixels as *N* x *N* rectangles (effectively lowering the window resolution). This "framezoom" extension, SGIX_framezoom, is available as a separate patch under both IRIX 5.3 and IRIX 6.2.

Caution: This extension is experimental. The interface and supported systems may change in the future.

When using the extension, consider the following performance tips:

- The extension works best when texturing is enabled. When pixels are zoomed up by *N*, you can expect the fill rate to go up by about $N^2/2$. This number is an estimate; a speed-up of this magnitude occurs only if texturing performance has been optimized as explained in the last bullet of "Getting the Optimum Fill Rates" on page 270.
- When texturing is not enabled, performance, although faster than the texture map case, is relatively slow compared to the non-framezoomed case. Actually, a framezoom value of 2 is slower than if framezoom was not enabled. The reason is that the graphics hardware in low-end systems is optimized for flat or interpolated spans, and not for cases where the color changes from pixel to pixel (as with texturing). When pixels are bigger (as with the framezoom extension), this benefit cannot be used.
- The framezoom factor can be changed on a frame-to-frame basis, so you can render with framezoom set to a larger value when you're moving around a scene, and lower the value, or turn framezoom off, when there are no changes in the scene.

For more detailed information, see the reference page for **glFrameZoomSGIX()** for those systems that have the patch installed.

Optimizing Performance on Mid-Range Systems

This section discusses optimizing performance for two of the Silicon Graphics mid-range systems: Elan™ graphics and Extreme™ graphics. For information on IMPACT systems, see "Optimizing Performance on IMPACT Systems" on page 276.

General Performance Tips

The following general performance tips apply to mid-range graphics systems:

- **Data size.** Mid-range graphics systems are optimized for word-sized and word-aligned data (one word is four bytes). Pixel read and draw operations are fast if the data is word aligned and each row is an integral number of words.
- **Extensions.** The following extensions are hardware accelerated:
  - "The Logical Operation Blending Extension" on page 143
  - "The Polygon Offset Extension" on page 145
  - "The ABGR Extension" on page 125

  Other available extensions are implemented in software.
- **Flushing.** Too many flushes, implicit or explicit, can adversely affect performance:
  - In single buffer mode, you may need to call **glFlush()** after the last of a series of primitives to force the primitives through the pipeline and expedite graphics processing (explicit flushing).
  - In double buffer mode, it is not necessary to call **glFlush()**; the **glXSwapBuffers()** call automatically flushes the pipeline (implicit flushing).

Optimizing Geometry Operations on Mid-Range Systems

Consider the following points when optimizing geometry operations for a mid-range system:

- **Antialiasing.** Mid-range graphics systems support hardware-accelerated lines of width 1.
- **Clipping.** Minimize clipping for better performance.

Optimizing Per-Fragment Operations on Mid-Range Systems

Consider the following issues when optimizing per-fragment operations for a mid-range system:

- **Alpha Blending.** Mid-range graphics systems support alpha 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.
- **Dithering.** 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. Disabling dither can improve the performance of **glClear()**. Dithering is enabled by default. To change that, call

  ```
  glDisable(GL_DITHER)
  ```

- **Fog.** Mid-range graphics systems do not accelerate per-fragment fog modes. To select a hardware-accelerated fog mode, call

  ```
  glHint (GL_FOG_HINT, GL_FASTEST)
  ```

- **Lighting.** Mid-range graphics systems accelerate all lighting features.
- **Pixel formats.** The GL_ABGR_EXT pixel format is much faster than the GL_RGBA pixel format. For details, see “The ABGR Extension” on page 125.

  The combinations of types and formats shown in Table 14-2 are the fastest.

Table 14-2 Pixel Formats and Types That Are Fast on Mid-Range Systems

Format	Type
GL_RGBA	GL_UNSIGNED_BYTE
GL_ABGR_EXT	GL_UNSIGNED_BYTE
GL_COLOR_INDEX	GL_UNSIGNED_SHORT
GL_COLOR_INDEX	GL_UNSIGNED_BYTE

- **Texture Mapping.** All texture mapping is performed in software. As a result, textured primitives run with reduced performance.

- Elan Graphics™ accelerates depth buffer operations on systems that have depth buffer hardware installed (default on Elan, optional on XS and XS24, not available on Entry systems).
- **Fast Clear Operations.** The hardware performs combined color and depth clear under the following conditions:
  – depth buffer is cleared to 1 and the depth test is GL_LEQUAL
  – depth buffer is cleared to 0 and the depth test is GL_GEQUAL

Optimizing Performance on IMPACT Systems

This section provides performance tips for IMPACT graphics systems. All information applies to both High IMPACT and Maximum IMPACT systems, unless specifically stated otherwise. You learn about:

- "General Tips for Performance Improvement"
- "Achieving Peak Geometry Performance"
- "Using Textures"
- "Using Images"
- "Accelerating Color Space Conversion"
- "Using Display Lists Effectively"
- "Offscreen Rendering Capabilities"

General Tips for Performance Improvement

This section provides some general tips for improving overall rendering performance. It also lists some features that are much faster than on previous systems and may now be used by applications that could not consider them before.

- **Fill-rate limited applications.** Because per-primitive operations (transformations, lighting, and so on) are very efficient on IMPACT systems, applications may find that they are fill-rate limited when drawing large polygons (more than 50 pixels per triangle). In that case, you can actually increase the complexity of per-primitive operations at no cost to overall performance. For example, additional lights or two-sided lighting may come for free.

For general advice on improving performance for fill-rate limited applications, see "Tuning the Raster Subsystem" on page 243. Specific for IMPACT systems is that texturing is optimized.

- **Geometry-limited applications.** For applications that draw many small polygons, consider a different approach: Use textures to avoid drawing so many triangles. See "Using Textures" on page 278.
- **Clipping.** For optimum performance, avoid clipping. Special hardware supports clipping within a small range outside of the viewport. By keeping geometry within this range, you may be able to significantly reduce clipping overhead.
- **GLU NURBS.** If you use GLU NURBS, store the tessellation result in display lists to take full advantage of evaluator performance. Don't for example, recompute tessellations.
- **Antialiasing.** Antialiased lines on IMPACT systems are high quality and fast. Applications that did not use antialiased lines before because of the performance penalty may now be able to take advantage of them. All antialiased lines are rendered with the same high quality, regardless of the settings of GL_LINE_SMOOTH_HINT. Although available, wide antialiased lines are not supported in hardware and should be avoided. Wide antialiased points are supported in hardware with good performance.

  Multisampling is not supported. Antialiasing of polygons is not supported in hardware. You can, however, draw antialiased line loops around polygons to get antialiasing.

Achieving Peak Geometry Performance

Because of the IMPACT hardware setup, rendering of primitives is especially fast if you follow these recommendations:

- **Triangles.** Work with triangle strips consisting of six triangles (or multiples of six). Render individual triangles in groups of four (or multiples or four).

  Note that the hardware allows mixing of different lengths of triangle strips. Grouping like primitives is highly recommended.
- **Quads.** Work with quad strips consisting of three quads (or multiples of three). Render individual quads in sets of three (or multiples of three).

- Use **glLoadIdentity()** to put identity matrixes on the stack. The system can optimize the pipeline if the identity matrix is used, but does not check whether a matrix loaded by **glLoadMatrix()** is the identity matrix.

Using Textures

IMPACT systems render textures quickly: Texturing is only slightly slower than non-textured fill rates and significantly improves image quality for your application. To get the most benefit from textures, use the extensions to OpenGL for texture management as follows:

- Use the texture object extension (EXT_texture_object) to keep as many textures resident in texture memory as possible. The extension lets you bind a texture to a name, then use it as needed (similar to the way you define and call a display list). The extension also allows you to specify a set of textures and prioritize which textures should be resident in texture memory. For more information, see “The Texture Object Extension” on page 93.
- Use the texture-LOD extension to minimize the number of mipmap levels you need to have resident while at the same time having the advantage of several levels of detail. For more information, see “The Texture LOD Extension” on page 121.
- Use the subtexture extension to make texture definition more efficient. For example, assume an application uses several large textures, all of the same size and component type. Instead of declaring multiple textures, declare one, then use **glTexSubImage2DEXT()** to redefine the image as needed. For more information, see “The Subtexture Extension” on page 96.
- Use the GL_RGBA4 internal format to improve performance and conserve memory (see Table 6-1, “Some Internal Formats Supported by the Texture Extension,” on page 87).

  This format is especially important if you have a large number of textures. The quality is reduced, but you can fit more textures into memory because they need less space.
- Use the GL_RGBA4 internal format and the packed pixels extension to minimize disk space and improve download rate (see “The Packed Pixels Extension” on page 125).
- Use the 3D texture extension for volume rendering. Note, however, that due to the large amount of data, you typically have to tile the texture. You can set up the

texture as a volume and slice through it as needed. For more information, see "The 3D Texture Extension" on page 100.

- If you use GL_LUMINANCE and GL_LUMINANCE_ALPHA textures, you can speed up loading by using the texture select extension.
- For IMPACT graphics, data coherence enhances performance. If you're drawing texture-mapped points, short lines, or very small triangles, try to draw together the ones that are close together in the image. Don't bounce around, or drawing may slow down.

Using Images

This section provides some tips for using images on IMPACT systems.

On many systems, a program encounters a noticeable performance cliff when a certain specific feature (for example depth-buffering) is turned on, or when the number of modes or components exceeds a certain limit.

On IMPACT systems, performance scales with the number of components. For example, on some systems, a switch from RGBA to RGB may not result in a change in performance, while on IMPACT systems, you should expect a performance improvement of 25%. (Note that while this applies to loading textures, it does not apply to using loaded textures.)

Here are some additional hints for optimizing image processing:

- Instead of **glPixelMaps()**, use the Silicon Graphics color table extension, discussed in "The Color Table Extension" on page 136, especially when working with GL_LUMINANCE or GL_LUMINANCE_ALPHA images.

  OpenGL requires expansion of pixels using formats other than GL_RGBA to GL_RGBA. Conceptually, this expansion takes place before any pixel operation is applied. IMPACT systems attempt to postpone expansion as long as possible: this improves performance (operations must be performed on all components present in an image—a non-expanded image has fewer components and therefore requires less computation). Because pixel maps are inherently four components, GL_LUMINANCE and GL_LUMINANCE_ALPHA images must be expanded (a different lookup table is applied to the red, green, and blue components derived from the luminance value). However, if the internal format of an image matches the internal format of the color table, IMPACT hardware postpones the expansion, which speeds up processing.

- The convolution extension, discussed in "The Convolution Extension" on page 129 has been optimized. If possible, use the extension with separable convolution filters.

  IMPACT systems are tuned for 3 x 3, 5 x 5, and 7 x 7 convolution kernels. If you choose a kernel size not in that set, performance is comparable to that of the closest member of the set. For example, if you specify 2 x 7, performance is similar to using 7 x 7.

- Use texture-based zooming instead of **glPixelZoom()**.

  Texture loading and interpolation is fast on IMPACT, and texture-based zooming therefore results in a speed increase and higher-quality, more controllable results.

- Where possible, minimize color table and histogram sizes and the number of color tables activated. If you don't, you may experience performance loss because the two compete for limited resources with other OpenGL applications.

Accelerating Color Space Conversion

IMPACT systems provide accelerated color space conversions and device-specific color matching.

- **Linear color space conversion.** Use the color matrix extension to handle linear color space conversion, such as CMY to RGB, in hardware. This extension is also useful for reassigning or duplicating components. See "The Color Matrix Extension" on page 128 for more information.
- **Non-linear color space conversions.** Use the 3D and 4D texture extension for color conversion (for example, RGBA to CMYK). Using the **glPixelTexGenSGIX()** command, you can direct pixels into the lookup table and get other pixels out. Performance has been optimized.

Using Display Lists Effectively

If you work on a CAD application, or other application that uses relatively static data, and therefore find it useful to use display lists instead of immediate mode, you can benefit from the display list implementation on IMPACT systems:

- When the display list is compiled, most OpenGL functions are stored in a format the hardware cannot use directly. At execution time, these display list segments are simply copied to the graphics hardware with little CPU overhead.

- A further optimization is that a DMA mechanism can be used for a subset of display lists. By default, the CPU feeds the called list to the graphics hardware. Using DMA display lists, the host gives up control of the bus and IMPACT uses DMA to feed the contents to the graphics pipeline. The speed improvement at the bus is fourfold; however, a setup cost makes this improvement irrelevant for very short lists. The break-even point varies depending on the list you're working with, whether it's embedded in other lists, and other factors.

Display List Compilation on IMPACT Hardware

The functions that are direct (use hardware formats) will change over time. The following items are currently NOT compiled to direct form:

- **glCallLists()** and **glListBase()**
- all imaging functions
- all texture functions, with the exception of **glBindTextureEXT()**
- **glHint()**, **glClear()**, and **glScissor()**
- **glEnable()** and **glDisable()**
- **glPushAttrib()** and **glPopAttrib()**
- all evaluator functions
- most OpenGL extensions

DMA Display Lists on IMPACT Systems

If a display list meets certain criteria, IMPACT systems use DMA to transfer data from the CPU to the graphics pipeline. This is useful if an application is bus limited. It can also be an advantage in a multi-threaded application, because the CPU can do some other work while the graphics subsystem pulls the display list over.

The DMA method is used under the following conditions:

- Only functions that are compiled down to direct form are used.
- There is no hierarchy in the display list that is more than eight levels deep.
- If the display list hierarchy uses texture objects, all textures that are referenced have to fit into hardware texture memory (TRAM) at the same time.

Note that the system tests recursively whether the DMA model is appropriate: If an embedded display list meets the criteria, it can be used in DMA mode even if the higher-level list is processed by the CPU.

Offscreen Rendering Capabilities

Offscreen rendering can be accelerated using the pixel buffer extension discussed in "The Pixel Buffer Extension" on page 179.

Optimizing Performance on RealityEngine Systems

This section provides information on optimizing applications for RealityEngine and RealityEngine2. It discusses:

- "Optimizing Geometry Performance"
- "Optimizing Rasterization"
- "Optimizing Use of the Vertex Array Extension"
- "Optimizing Multisampling and Transparency"
- "Optimizing the Imaging Pipeline"

Optimizing Geometry Performance

Here are some tips for improving RealityEngine geometry performance:

- **Primitive length.** Most systems have a characteristic primitive length that the system is optimized for. On RealityEngine systems, multiples of 3 vertices are preferred, and 12 vertices (for example a triangle strip that consists of 10 triangles) result in the best performance.
- **Fast mode changes.** Changes involving logic op, depth func, alpha func, shade model, cullface, or matrix mode are fast.
- **Slow mode changes**. Changes involving texture binding, lighting and material changes, line width and point size changes, scissor, or viewport are slow.
- **Texture coordinates.** Automatic texture coordinate generation with **glTexGen()** results in a relatively small performance degradation.

- **Quads and polygons.** When rendering quads, use GL_POLYGON instead of GL_QUADS. The GL_QUADS primitive checks for self-intersecting quads and is therefore slower.

Optimizing Rasterization

This section discusses optimizing rasterization. While it points out a few things to watch out for, it also provides information on features that were expensive on other systems but are acceptable on RealityEngine systems:

- After a clear (or a command to fill a large polygon), send primitives to the geometry engine for processing. Geometry can be prepared as the clear or fill operations take place.
- Texturing is free on a RealityEngine if you use a 16-bit texel internal texture format. There are 16-bit texel formats for each number of components. Using a 32-bit texel format yields half the fill rate of the 16-bit texel formats. See “The Texture Extension” on page 85 for more information on internal formats.
- The use of detail texture and sharpen texture usually incurs no additional cost and can greatly improve image quality. Note, however, that texture management can become expensive if a detail texture is applied to many base textures. Use detail texture but keep detail and base paired and detail only a few base textures. See “The Sharpen Texture Extension” on page 107 and “The Detail Texture Extension” on page 112.
- If textures are changing frequently, use the subtexture extension to incrementally load texture data. RealityEngine systems are optimized for 32 x 32 subimages. See “The Subtexture Extension” on page 96.
- There is no penalty for using the highest-quality mipmap filter (GL_LINEAR_MIPMAP_LINEAR) if 16-bit texels are used (for example, the GL)RGBA4_EXT internal format). See “The Texture Extension” on page 85.
- Local lighting or multiple lights are possible without an unacceptable degradation in performance. As you turn on more lights, performance degrades slowly.
- Simultaneous clear of depth and color buffers is optimized in hardware.
- Antialiased lines and points are hardware accelerated.

Optimizing Use of the Vertex Array Extension

If you use the vertex array extension (“The Vertex Array Extension” on page 151), the following cases are currently accelerated for RealityEngine (each line corresponds to a different special case). To get the accelerated routine, you need to make sure your vertices correspond to the given format by using the correct size and type in your enable routines, and also by enabling the proper arrays:

- Vertex2f
- Normal3f Vertex3f
- Color3f Vertex3f
- Color4f Vertex3f
- Normal3f Vertex3f
- TexCoord2f Vertex3f
- Color4f TexCoord2f Vertex3f
- Color3f Normal3f Vertex3f
- Color4f Normal3f Vertex3f
- Normal3f TexCoord2f Vertex3f
- Color4f TexCoord2f Normal3f Vertex3f

Optimizing Multisampling and Transparency

Multisampling provides full-scene antialiasing with performance sufficient for a real-time visual simulation application. However, it isn’t free and it adds to the cost of some fill operations. With RealityEngine graphics, some fragment processing operations (blending, depth buffering, stenciling) are essentially free if you aren’t multisampling, but do reduce performance if you use a multisample-capable visual. Texturing is an example of a fill operation that can be free on a RealityEngine and isn’t affected by the use of multisampling. Note that when using a multisample-capable visual, you pay the cost even if you disable multisampling.

Below are guidelines for optimizing performance for multisampling:

- Multisampling offers an additional performance optimization that helps balance its cost: a virtually free screen clear. Technically, this operation doesn’t really clear the screen, but rather allows you to set the depth values in the framebuffer to be undefined. Therefore, use of this clear requires that every pixel in the window be rendered every frame. This clear is invoked with **glTagSampleBufferSGIX()** (see the reference page for more information).

- When multisampling, using a smaller number of samples and color resolution results in better performance. 8 samples with 8-bit RGB components and a 24-bit depth buffer usually result in good performance and quality. (32-bit depth buffers are rarely needed.)
- Multisampling with stencilling is expensive. If it becomes too expensive, use the polygon offset extension to deal with decal tasks (for example runway strips). See "The Polygon Offset Extension" on page 145.
- There are two ways of achieving transparency on a RealityEngine system: alpha-blending and subpixel screen-door transparency using **glSampleMaskSGIS().** Alpha blending may be slower, since more buffer memory may need to be accessed. For more information about screen-door transparency, see "The Multisample Extension" on page 154.

Optimizing the Imaging Pipeline

Here are some points that help you optimize the imaging pipeline:

- Unsigned color types are faster than signed or float types.
- Smaller component types (for example, GL_UNSIGNED_BYTE) require less bandwidth from the host to the graphics pipeline and are faster than larger types.
- The slow pixel drawing path is used when fragment operations (depth or alpha testing, and so on) are used, or when the format is GL_DEPTH_COMPONENT, or when multisampling is enabled and the visual has a multisample buffer.

Using the Color Matrix and the Color Writemask

Your application might perform RGBA imaging operations (for example convolution, histogram, and such) on a single-component basis. This is the case either when processing gray scale (monochrome) images, or when different color components are processed differently.

RealityEngine systems currently do not support RGBA-capable monochrome visuals (a feature that is introduced by the framebuffer configuration extension; see "The Framebuffer Configuration Extension" on page 171). You must therefore use a four-component RGBA visual even when performing monochrome processing. Even when monochrome RGBA-capable visuals are supported, you may find it beneficial to use four-component visuals in some cases, depending on your application, to avoid the overhead of the **glXMakeCurrent()** or **glXMakeCurrentReadSGI()** call.

On RealityEngine systems, monochrome imaging pipeline operations are about four times as fast as the four-component processing. This is because only a quarter of the data has to be processed or transported either from the host to graphics subsystem—for example, for **glDrawPixels()**—or from the framebuffer to the graphics engines—for example, for **glCopyPixels().**

The RealityEngine implementation detects monochrome processing by examining the color matrix (see "Tuning the Imaging Pipeline" on page 246) and the color writemask.

The following operations are optimized under the set of circumstances listed below:

- **glDrawPixels()** with convolution enabled and
  - the pixel format is GL_LUMINANCE or GL_LUMINANCE_ALPHA
  - the color matrix is such that the active source component is red
- **glCopyPixels()** and the absolute value of GL_ZOOM_X and GL_ZOOM_Y is 1.

Here's the set of circumstances to be met:

- All pixel maps and fragment operations are disabled.
- The color matrix does not scale any of the components.
- The post color matrix scales and biases for all components are 1 and 0, respectively.
- Either write is enabled only for a single component (R, G, B, or A), or alpha-component write is disabled.

Appendix A

OpenGL and IRIS GL

The first step in porting a RealityEngine IRIS GL application to OpenGL is to consult the *OpenGL Porting Guide*. It covers all the core IRIS GL and OpenGL functionality, window and event handling, and OpenGL extensions up to and including those in IRIX 5.3.

This appendix provides some additional information about porting IRIS GL to OpenGL, pointing to the extensions discussed in earlier chapters of this book where appropriate. In order to make it as short and comprehensible as possible, it won't repeat any information from the *OpenGL Porting Guide*.

Some IRIS GL Functionality and OpenGL Equivalents

This section provides an alphabetical list of IRIS GL functions and some other functionality and either a pointer to related OpenGL functions or an explanation of how to implement similar functionality in OpenGL.

backbuffer, frontbuffer

The framebuffer update semantics for rendering into multiple color buffers are different in IRIS GL and OpenGL. OpenGL on RealityEngine systems actually implements the IRIS GL semantics (computing one color value and writing it to all buffers) rather than the correct OpenGL semantics (computing a separate color value for each buffer). This can cause unexpected results if blending is used.

blendcolor

See "Blending Extensions" on page 140.

blendfunction

See "Blending Extensions" on page 140.

convolve

See “The Convolution Extension” on page 129.

displacepolygon

The OpenGL equivalent, **glPolygonOffsetEXT()**, is more general than **displacepolygon()**. You may need to tweak the parameter values to get the proper results. See “The Polygon Offset Extension” on page 145.

dither

OpenGL provides no control over video dithering. (This is also the case for IRIS GL in IRIX 5.3, unless overridden by an environment variable.)

fbsubtexload

See “The Subtexture Extension” on page 96.

gamma

Use the XSGIvc extension. See “Stereo Rendering” on page 74.

glcompat GLC_SET_VSYNC, GLC_GET_VSYNC, GLC_VSYNC_SLEEP

For GLC_GET_VSYNC, use **glXGetVideoSyncSGI()**. For GLC_VSYNC_SLEEP, use **glXWaitVideoSyncSGI()**. See “The Video Synchronization Extension” on page 167.

GLC_SET_VSYNC has no equivalent in OpenGL. To replace it, maintain a sync counter offset in a static variable.

glcompat SIR_VEN_INTERFACE, SIR_VEN_VIDEOCOPY

(This function copies Sirius video to the framebuffer)

Supported, with some constraints. Use **glXCreateGLXVideoSourceSGIX()** to create a video source context, **glXMakeCurrentReadSGI()** to set up the video source for a data transfer, and **glCopyPixels()** to copy the video data to the framebuffer.

hgram, gethgram (histogram)

Supported for:

```
glDrawPixels(lrectwrite), glCopyPixels(rectcopy),
glReadPixels(lrectread)
```

Use **glGetHistogramEXT()** and **glHistogramEXT()**; see “The Histogram and Minmax Extensions” on page 132.

ilbuffer, ildraw, readsource(SRC_ILBUFFER)

(This function provides accelerated drawing to offscreen framebuffer memory.)

See “The Pixel Buffer Extension” on page 179.

istexloaded

Use **glAreTexturesResidentEXT()**; see “Texture Priorities and Residency” on page 95 for texture management.

leftbuffer, rightbuffer

Use **glXChooseVisual()** and **glDrawBuffer()** for stereo in a window. For old-style stereo, see **XSGISetStereoMode()**.

libsphere—sphdraw, sphgnpolys, sphfree, sphmode, sphobj, sphrotmatrix, sphbgnbitmap, sphendbitmap, sphcolor

gluSphere() provides polygonal spheres. Only bilinear tessellation is supported; octahedral, icosahedral, barycentric, and cubic tessellations are not supported.

There is no support for the canonical orientation capability (sphrotmatrix), hemispheres (SPH_HEMI), or bitmap spheres.

linesmooth

Antialiased lines are supported, with one caveat: it is not possible to draw blended antialiased lines in a multisampled window, even when multisampling is disabled. See **glHint()** and **glEnable()** with the GL_LINE_SMOOTH parameter.

minmax, getminmax

For minimum and maximum pixel values, use **glGetMinmaxEXT()** and **glMinmaxEXT()**; see "The Histogram and Minmax Extensions" on page 132.

mswapbuffers

Not supported.

Swapping multiple windows (for example, main and overlay buffers, or windows on genlocked pipes) from multiple threads can be accomplished fairly reliably with semaphored calls to **glXSwapBuffers()**. The following code fragment outlines the approach:

```
/* Create some semaphores: */
       usptr_t* arena = usinit("/usr/tmp/our_arena");
       usema_t* pipe0ready = usnewsema(arena, 0);
       usema_t* pipe1ready = usnewsema(arena, 0);

/* After the process for pipe0 finishes its frame, it signals its
completion and waits for pipe1. When pipe1 is also ready, pipe0 swaps:
*/
       usvsema(pipe0ready);
       uspsema(pipe1ready);
       glXSwapBuffers(dpy, drawable);

/* The process for pipe 1 does the converse: */
       usvsema(pipe1ready);
       uspsema(pipe0ready);
       glXSwapBuffers(dpy, drawable);
```

multisample, getmultisample, msalpha, msmask, mspattern, mssize (multisample antialiasing)

Supported. See "The Multisample Extension" on page 154.

For msalpha, see **glEnable()** with arguments GL_SAMPLE_ALPHA_TO_MASK_SGIS and GL_SAMPLE_ALPHA_TO_ONE_SGIS.

For msmask, see **glSampleMaskSGIS()**.

For mspattern, see **glSamplePatternSGIS()**.

For mssize, see **glXChooseVisual()**.

For “light points,” use multisampling with

```
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST)
```

The maximum point diameter is 3 (the same as IRIS GL).

For fast tag clear, see **glTagSampleBufferSGIX()**.

pixelmap

Differs from IRIS GL. The OpenGL function **glPixelMap()** specifies a lookup table for pixel transfer operations, just as pixelmap does in IRIS GL. However, the location of the lookup table in the pixel processing pipeline is different. The IRIS GL lookup table appears immediately after convolution, while the OpenGL lookup table appears almost at the beginning of the pipeline (immediately after the first scale and bias). The two pipelines are equivalent only when convolution is disabled.

Pixel mapping is supported for the following calls:

```
glDrawPixels(lrectwrite), glCopyPixels(rectcopy),
glReadPixels(lrectread), glTexImage(texdef),
```

It is not supported for

```
glTexSubImage(subtexload)
```

pixmode

Most of the functions of pixmode are supported, albeit in different ways:

- PM_SHIFT, PM_ADD24: Use the OpenGL color matrix extension to swizzle color components or to scale and bias pixel values. See **glPixelTransfer()**.
- PM_EXPAND, PM_C0, PM_C1: Use the standard OpenGL color lookup table to convert bitmap data to RGBA. See **glPixelTransfer()** and **glPixelMap()**.
- PM_TTOB, PM_RTOL: Use **glPixelZoom()** with negative zoom factors to reflect images when drawing or copying. Reflection during reading is not supported.
- PM_INPUT_FORMAT, PM_OUTPUT_FORMAT, PM_INPUT_TYPE, PM_OUTPUT_TYPE, PM_ZDATA: Use the **glReadPixels()**, **glDrawPixels()**, and **glCopyPixels()** type and format parameters.
- PM_OFFSET, PM_STRIDE, PM_SIZE: Use **glPixelStore()**.

pntsize

Supported. See comments under "Some IRIS GL Functionality and OpenGL Equivalents" on page 290.

polymode

OpenGL doesn't support PYM_HOLLOW. **glPolygonMode(GL_LINE)** is the closest approximation. See also **glPolygonOffsetEXT()** and "The Polygon Offset Extension" on page 145.

polysmooth

OpenGL doesn't support PYM_SHRINK.

popup planes

OpenGL doesn't support drawing in the popup planes.

readcomponent

Use the color matrix extension (see **glPixelTransfer()**) to extract one or more color channels for image processing. The implementation is optimized for the case in which all channels but one are multiplied by zero, and all framebuffer channels but one are write-masked (see **glColorMask()**). See "Using the Color Matrix and the Color Writemask" on page 285.

The color matrix is supported for **glDrawPixels()** and **glCopyPixels()** but not for **glTexImage()** or **glReadPixels()**.

See "The Color Matrix Extension" on page 128 for more information.

RGBwritemask

OpenGL supports masking an entire RGBA color channel, but not arbitrary sets of bits within an RGBA color channel.

setvideo, setmonitor

OpenGL has no support for these routines.

Video output format should be changed with the *setmon* command (this is now recommended for IRIS GL as well as OpenGL).

OpenGL supports stereo-in-a-window; see **glXChooseVisual()** and **glDrawBuffer()**. For old-style stereo, see **XSGISetStereoMode()**.

Use the Video Library (VL) or the XSGIvc extension for other video device control tasks.

subtexload

See “The Subtexture Extension” on page 96.

tevdef, tevbind

TV_COMPONENT_SELECT (the ability to pack multiple shallow textures together, then unpack and select one of them during drawing) is not supported.

texbind

Texture definition and binding are combined into a single operation in standard OpenGL. However, the texture object extension makes them separate again (albeit in a manner that differs from IRIS GL). Use **glBindTextureEXT()**; see “The Texture Object Extension” on page 93.

Detail texturing (TX_TEXTURE_DETAIL) is supported. Use **glDetailTexFuncSGIS()**; see “The Detail Texture Extension” on page 112.

Simple texture memory management (TX_TEXTURE_IDLE) is supported. Use **glPrioritizeTexturesEXT()**; see “Texture Priorities and Residency” on page 95.

texdef

1D, 2D, and 3D textures are supported. See **glTexImage1D()**, **glTexImage2D()**, and **glTexImage3DEXT()**; see “The 3D Texture Extension” on page 100.

TX_FAST_DEFINE is not supported. A copy of each texture is always placed in kernel memory for graphics context switching. (OpenGL semantics require the graphics library to read the texture exactly once; it cannot re-read the texture later to refresh texture memory after a graphics context switch). Loading subtextures is still possible, however; use **glTexSubImage2DEXT()** (see “The Subtexture Extension” on page 96).

The TX_BILINEAR_LEQUAL and TX_BILINEAR_GEQUAL filtering options, which are used to implement shadows, are not supported.

The TX_BICUBIC filter option, which is used for bicubic filtering and as a workaround for the lack of point sampling, is also not supported.

The TX_MINFILTER options for mipmapping are supported for 1D and 2D textures, but not for 3D textures. 3D textures must use GL_NEAREST (TX_POINT) or GL_LINEAR (TX_BILINEAR) filtering modes.

OpenGL differs from IRIS GL in that filtered versions of the texture image (when required by the current minification filter) are not generated automatically; the application must load them explicitly. Thus the TX_MIPMAP_FILTER_KERNEL token is not supported.

Separate magnification filters for color and alpha (TX_MAGFILTER_COLOR and TX_MAGFILTER_ALPHA) are not supported in the general case. However, it is possible to specify separate alpha and color magnification filters for detail and sharp texturing. See **glTexParameter()**.

Sharp texture filtering (TX_SHARPEN) is supported. Use **glTexParameter()** for setting the filtering mode, and **glSharpenTexFuncSGIS()** for setting the scaling function (TX_CONTROL_POINT, TX_CONTROL_CLAMP). See “The Sharpen Texture Extension” on page 107.

Detail texture (TX_ADD_DETAIL and TX_MODULATE_DETAIL) is supported for 2D. The parameters are specified differently from those in IRIS GL. See **glTexParameter()** for setting the filtering mode, and **glDetailTexFuncSGIS()** for setting the scaling function (TX_CONTROL_POINT, TX_CONTROL_CLAMP). See “The Detail Texture Extension” on page 112.

The TX_WRAP mode TX_SELECT is not supported. OpenGL provides GL_CLAMP (TX_CLAMP) and GL_REPEAT (TX_REPEAT).

TX_INTERNAL_FORMAT and all IRIS GL texel internal formats are supported. See the *components* parameter of **glTexImage2D()** for a list of the OpenGL internal formats. See also “The Texture Extension” on page 85 for new internal formats and how to use them.

TX_TILE (multipass rendering for high-resolution textures) is not supported directly. OpenGL border clamping can emulate tiling if you use the edges of neighboring tiles as the borders for the current tile.

tlutbind

In OpenGL, tlut definition and binding are combined into a single operation, and tluts apply to all texturing (rather than being bound to a particular texture target). See the comments under tlutdef.

tlutdef

Use **glTexColorTableParameterSGI()** for a description of the OpenGL texture color lookup process; see "The Texture Color Table Extension" on page 105. Use **glColorTableSGI()** for information about loading the lookup table; see "The Color Table Extension" on page 136.

The OpenGL lookup table semantics differ from those that IRIS GL used.

- The case described in the **tlutdef()** reference page and shown in the following table cannot be emulated in OpenGL. (nc stands for number of components, I for intensity, A for Alpha, R, G, and B for Red, Green, and Blue.)

tlut nc	texture nc	action
4	3	R, G, B, B looks up R, G, B, A

- The cases shown in the following table are supported directly, or can be supported with a judicious choice of table values and calls to **glEnable()**.

tlut nc	texture nc	action
2	1	I looks up I,A
	2	I,A looks up I,A
	3	R,G,B pass unchanged
	4	R,G,B,A pass unchanged
3	1	I looks up R,G,B
	2	I,A pass unchanged
	3	R,G,B looks up R,G,B
	4	R,G,B,A pass unchanged

tlut nc	texture nc	action
4	1	I looks up R,G,B,A
	2	I looks up RGB; A looks up A
	4	R,G,B,A looks up R,G,B,A

OpenGL supports cases that are not available under IRIS GL. See **glTexColorTableParameterSGI()** for more information.

underlays

There are no X11 Visuals for the underlay planes, so OpenGL rendering to underlays is not supported.

Benchmarks

This appendix contains a sample program you can use to measure the performance of an OpenGL operation. For an example of how the program can be used with a small graphics applications, see Chapter 13, “Tuning Graphics Applications: Examples.”

```
/**********************************************************************
 * perf - framework for measuring performance of an OpenGL operation
 *
 * Compile with: cc -o perf -O perf.c -lGL -lX11
 *
**********************************************************************/


#include <GL/glx.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <sys/time.h>


char* ApplicationName;
double Overhead = 0.0;
int VisualAttributes[] = { GLX_RGBA, None };
int WindowWidth;
int WindowHeight;


/**********************************************************************
 * GetClock - get current time (expressed in seconds)
**********************************************************************/
double
GetClock(void) {
        struct timeval t;

        gettimeofday(&t);
        return (double) t.tv_sec + (double) t.tv_usec * 1E-6;
        }
```

```
/**********************************************************************
 * ChooseRunTime - select an appropriate runtime for benchmarking
**********************************************************************/
double
ChooseRunTime(void) {
        double start;
        double finish;
        double runTime;

        start = GetClock();

        /* Wait for next tick: */
        while ((finish = GetClock()) == start)
                ;

        /* Run for 100 ticks, clamped to [0.5 sec, 5.0 sec]: */
        runTime = 100.0 * (finish - start);
        if (runTime < 0.5)
                runTime = 0.5;
        else if (runTime > 5.0)
                runTime = 5.0;

        return runTime;
        }

/**********************************************************************
 * FinishDrawing - wait for the graphics pipe to go idle
 *
 * This is needed to make sure we're not including time from some
 * previous uncompleted operation in our measurements.  (It's not
 * foolproof, since we can't eliminate context switches, but we can
 * assume our caller has taken care of that problem.)
**********************************************************************/
void
FinishDrawing(void) {
        glFinish();
        }

/**********************************************************************
 * WaitForTick - wait for beginning of next system clock tick; return
 * the time
**********************************************************************/
```

```
double
WaitForTick(void) {
        double start;
        double current;

        start = GetClock();

        /* Wait for next tick: */
        while ((current = GetClock()) == start)
                ;

        /* Start timing: */
        return current;
        }

/**********************************************************************
 * InitBenchmark - measure benchmarking overhead
 *
 * This should be done once before each risky change in the
 * benchmarking environment.  A ``risky'' change is one that might
 * reasonably be expected to affect benchmarking overhead.  (For
 * example, changing from a direct rendering context to an indirect
 * rendering context.)  If all measurements are being made on a single
 * rendering context, one call should suffice.
 **********************************************************************/
void
InitBenchmark(void) {
        double runTime;
        long reps;
        double start;
        double finish;
        double current;

        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /* Wait for the pipe to clear: */
        FinishDrawing();

        /* Measure approximate overhead for finalization and timing
         * routines
         */
        reps = 0;
        start = WaitForTick();
        finish = start + runTime;
```

```
        do {
                FinishDrawing();
                ++reps;
                } while ((current = GetClock()) < finish);

        /* Save the overhead for use by Benchmark(): */
        Overhead = (current - start) / (double) reps;
        }

/***********************************************************************
 * Benchmark - measure number of caller's operations performed per
 * second.
 * Assumes InitBenchmark() has been called previously, to initialize
 * the estimate for timing overhead.
 ***********************************************************************/
double
Benchmark(void (*operation)(void)) {
        double runTime;
        long reps;
        long newReps;
        long i;
        double start;
        double current;

        if (!operation)
                return 0.0;

        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /*
         * Measure successively larger batches of operations until we
         * find one that's long enough to meet our runtime target:
         */
        reps = 1;
        for (;;) {
                /* Run a batch: */
                FinishDrawing();
                start = WaitForTick();
                for (i = reps; i > 0; --i)
                        (*operation)();
                FinishDrawing();
```

```
                /* If we reached our target, bail out of the loop: */
                current = GetClock();
                if (current >= start + runTime + Overhead)
                        break;

                /*
                 * Otherwise, increase the rep count and try to reach
                 * the target on the next attempt:
                 */
                if (current > start)
                        newReps = reps *
                                (0.5 + runTime / (current - start -
                                                         Overhead));
                else
                        newReps = reps * 2;
                if (newReps == reps)
                        reps += 1;
                else
                        reps = newReps;
                }

        /* Subtract overhead and return the final operation rate: */
        return (double) reps / (current - start - Overhead);
        }

/**********************************************************************
 * Test - the operation to be measured
 *
 * Will be run several times in order to generate a reasonably accurate
 * result.
**********************************************************************/
void
Test(void) {
        /* Replace this code with the operation you want to measure: */
        glColor3f(1.0, 1.0, 0.0);
        glRecti(0, 0, 32, 32);
        }

/**********************************************************************
 * RunTest - initialize the rendering context and run the test
**********************************************************************/
void
RunTest(void) {
        if (Overhead == 0.0)
                InitBenchmark();
```

```
        /* Replace this sample with initialization for your test: */

        glClearColor(0.5, 0.5, 0.5, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, WindowWidth, 0.0, WindowHeight, -1.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        printf("%.2f operations per second\n", Benchmark(Test));
        }

/**********************************************************************
 * ProcessEvents - handle X11 events directed to our window
 *
 * Run the measurement each time we receive an expose event.
 * Exit when we receive a keypress of the Escape key.
 * Adjust the viewport and projection transformations when the window
 * changes size.
 *********************************************************************/
void
ProcessEvents(Display* dpy) {
        XEvent event;
        Bool redraw = 0;

        do {
                char buf[31];
                KeySym keysym;

                XNextEvent(dpy, &event);
                switch(event.type) {
                        case Expose:
                                redraw = 1;
                                break;
                        case ConfigureNotify:
                                glViewport(0, 0,
                                        WindowWidth =
                                                event.xconfigure.width,
                                        WindowHeight =
                                              event.xconfigure.height);
                                redraw = 1;
                                break;
```

```
                        case KeyPress:
                                (void) XLookupString(&event.xkey, buf,
                                        sizeof(buf), &keysym, NULL);
                                switch (keysym) {
                                        case XK_Escape:
                                                exit(EXIT_SUCCESS);
                                        default:
                                                break;
                                        }
                                break;
                        default:
                                break;
                        }
                } while (XPending(dpy));

        if (redraw) RunTest();
        }

/**********************************************************************
 * Error - print an error message, then exit
 **********************************************************************/
void
Error(const char* format, ...) {
        va_list args;

        fprintf(stderr, "%s:  ", ApplicationName);

        va_start(args, format);
        vfprintf(stderr, format, args);
        va_end(args);

        exit(EXIT_FAILURE);
        }

/**********************************************************************
 * main - create window and context, then pass control to ProcessEvents
 **********************************************************************/
int
main(int argc, char* argv[]) {
        Display *dpy;
        XVisualInfo *vi;
        XSetWindowAttributes swa;
        Window win;
        GLXContext cx;
```

```
ApplicationName = argv[0];

/* Get a connection: */
dpy = XOpenDisplay(NULL);
if (!dpy) Error("can't open display");

/* Get an appropriate visual: */
vi = glXChooseVisual(dpy, DefaultScreen(dpy),VisualAttributes);
if (!vi) Error("no suitable visual");

/* Create a GLX context: */
cx = glXCreateContext(dpy, vi, 0, GL_TRUE);

/* Create a color map: */
swa.colormap = XCreateColormap(dpy, RootWindow(dpy,
                           vi->screen), vi->visual, AllocNone);

/* Create a window: */
swa.border_pixel = 0;
swa.event_mask = ExposureMask | StructureNotifyMask |
                                                  KeyPressMask;
win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0,
                300, 300, 0,vi->depth, InputOutput, vi->visual,
                CWBorderPixel|CWColormap|CWEventMask, &swa);
XStoreName(dpy, win, "perf");
XMapWindow(dpy, win);

/* Connect the context to the window: */
glXMakeCurrent(dpy, win, cx);

/* Handle events: */
while (1) ProcessEvents(dpy);
}
```

Benchmarking Libraries: libpdb and libisfast

When optimizing an OpenGL application, there are two problems you need to address:

- When you're writing an OpenGL application, it's difficult to know whether a particular feature (like depth buffering or texture mapping) is fast enough to be useful.
- If you want your application to run fast on a variety of machines, while taking advantage of as many hardware features as possible, you need to write code that makes configuration decisions at runtime.

For the OpenGL predecessor IRIS GL, you could call **getgdesc()** to determine whether a feature had hardware support. For example, you could determine whether a Z buffer existed. If it did, you might assume that Z buffering was fast, and therefore your application would use it.

In OpenGL, things are more complicated. All the core features are provided, even when there is no hardware support for them and they must be implemented completely in software. There is no OpenGL routine that reports whether a feature is implemented partly or completely in hardware.

Furthermore, features interact in unpredictable ways. For example, a machine might have hardware support for depth buffering, but only for some comparison functions. Or depth buffering might be fast only as long as stencilling is not enabled. Or depth buffering might be fast when drawing to a window, but slow when drawing to a pixmap. And so on. A routine that identifies hardware support for particular features is actually a lot more complicated and less useful than you might think.

To decide whether a given OpenGL feature is fast, you have to measure it. Since the performance of a section of graphics code is dependent on many pieces of information from the runtime environment, no other method is as well-defined and reliable.

Keep in mind that while the results of the libisfast routines are interesting, they apply to limited special cases. Always consider using a more general tool like Open Inventor or IRIS Performer.

Performance measurement can be tricky:

- You need to handle the cases when you're displaying over a network, as well as locally.
- Think about flushing the graphics pipeline properly, and accounting for the resulting overhead.
- Measuring all the features needed by your application may take a while. Save performance measurements and reuse them whenever possible; users won't want to wait for measurements each time the application starts.
- Consider measuring things other than graphics: Disk and network throughput, processing time for a particular set of data, performance on uniprocessor and multiprocessor systems.

Libraries for Benchmarking

This appendix describes two libraries that can help with all of the tasks just mentioned:

- libpdb (**P**erformance **D**ata**B**ase). Routines for measuring execution rates and maintaining a simple database.
- libisfast. A set of routines demonstrating libpdb that answer common questions about the performance of OpenGL features (using reasonable but subjective criteria).

These libraries can't substitute for comprehensive benchmarking and performance analysis, and don't replace more sophisticated tools (like IRIS Performer and IRIS Inventor) that optimize application performance in a variety of ways. However, they can handle simple tasks easily.

Using libpdb

libpdb provides five routines:

- **pdbOpen()** opens the performance database.
- **pdbReadRate()** reads the execution rate for a given benchmark (identified by a machine name, application name, benchmark name, and version string) from the database.

- **pdbMeasureRate()** measures the execution rate for a given operation.
- **pdbWriteRate()** writes the execution rate for a given benchmark into the database.
- **pdbClose()** closes the performance database and writes it back to disk if necessary.

All libpdb routines return a value of type pdbStatusT, which is a bitmask of error conditions. If the value is zero (PDB_NO_ERROR), the call completed successfully. If the value is nonzero, it is a combination of one or more of the conditions listed in Table C-1:

Table C-1 Errors Returned by libpdb Routines

Error	Meaning
PDB_OUT_OF_MEMORY	Attempt to allocate memory failed.
PDB_SYNTAX_ERROR	Database contains one or more records that could not be parsed.
PDB_NOT_FOUND	Database does not contain the record requested by the application.
PDB_CANT_WRITE	Database file could not be updated.
PDB_NOT_OPEN	**pdbOpen()** was not invoked before calling one of the other libpdb routines.
PDB_ALREADY_OPEN	**pdbOpen()** was called while the database is still open (e.g., before **pdbClose()** is invoked).

Every program must call **pdbOpen()** before using the database, and **pdbClose()** when the database is no longer needed. **pdbOpen()** opens the database file (stored in $HOME/.pdb2 on UNIX systems) and reads all the performance measurements into main memory. **pdbClose()** releases all memory used by the library, and writes the database back to its file if any changes have been made by invoking **pdbWriteRate()**.

```
pdbStatusT pdbOpen(void);
pdbStatusT pdbClose(void);
```

pdbOpen() returns

- PDB_NO_ERROR on success
- PDB_OUT_OF_MEMORY if there was insufficient main memory to store the entire database
- PDB_SYNTAX_ERROR if the contents of the database could not be parsed or seemed implausible (for example a nonpositive performance measurement)

- PDB_ALREADY_OPEN if the database has been opened by a previous call to **pdbOpen()** and not closed by a call to **pdbClose()**

pdbClose() returns

- PDB_NO_ERROR on success
- PDB_CANT_WRITE if the database file is unwritable for any reason
- PDB_NOT_OPEN if the database is not open

Normally applications should look for the performance data they need before going to the trouble of taking measurements. **pdbReadRate()**, which is used for this, has the following prototype:

```
pdbStatusT pdbReadRate (const char* machineName,const char* appName,
          const char* benchmarkName,const char* versionString, double* rate )
```

machineName A zero-terminated string giving the name of the machine for which the measurement is sought. If NULL, the default machine name is used. (In X11 environments, the display name is an appropriate choice, and the default machine name is the content of the DISPLAY environment variable.)

appName Name of the application. This is used as an additional database key to reduce accidental collisions between benchmark names.

benchmarkName Name of the benchmark.

versionString The fourth argument is a string identifying the desired version of the benchmark. For OpenGL performance measurements, the string returned by **glGetString(GL_VERSION)** is a good value for this argument. Other applications might use the version number of the benchmark, rather than the version number of the system under test.

rate A pointer to a double-precision floating-point variable that receives the performance measurement (the "rate") from the database. The rate indicates the number of benchmark operations per second that were measured on a previous run. If **pdbReadRate()** returns zero, then it completed successfully and the rate is returned in the last argument. If the requested benchmark is not present in the database, it returns PDB_NOT_FOUND. Finally, if **pdbReadRate()** is called when the database has not been opened by **pdbOpen()**, it returns PDB_NOT_OPEN.

Example for pdbRead

```
main() {
        double rate;
        pdbOpen();
        if (pdbReadRate(NULL, "myApp", "triangles",
           glGetString(GL_VERSION), &rate)
               == PDB_NO_ERROR)
        printf("%g triangle calls per second\n", rate);
        pdbClose();
        }
```

When the application is run for the first time, or when the performance database file has been removed (perhaps to allow a fresh start after a hardware upgrade), **pdbReadRate()** is not able to find the desired benchmark. If this happens, the application should use **pdbMeasureRate()**, which has the following prototype, to make a measurement:

```
pdbStatusT pdbMeasureRate (pdbCallbackT initialize, pdbCallbackT operation,
                           pdbCallbackT finalize, int calibrate, double* rate)
```

initialize — A pointer to the initialization function. The initialization function is run before each set of operations. For OpenGL performance measurement, it's appropriate to use **glFinish()** for initialization, to make sure that the graphics pipe is quiet. However, for other performance measurements, the initialization function can create test data, preload caches, and so on. May be NULL, in which case no initialization is performed.

operation — A pointer to the operation function. This function performs the operations that are to be measured. Usually you'll want to make sure that any global state needed by the operation is set up before calling the operation function, so that you don't include the cost of the setup operations in the measurement.

finalize — A pointer to a finalization function. This is run once, after all the calls to the operation function are complete. In the example above, **glFinish()** ensures that the graphics pipeline is idle. It may be NULL, in which case no finalization is performed. The finalization function must be calibrated so that the overhead of calling it may be subtracted from the time used by the operation function. If the fourth argument is nonzero, then **pdbMeasureRate()** calibrates the finalization function. If the fourth argument is zero, then **pdbMeasureRate()** uses the results of the previous calibration. Recalibrating each measurement is the safest approach, but it roughly doubles the amount of time needed for a measurement. For OpenGL, it should be acceptable to calibrate once and recalibrate only when using a different X11 display.

rate A pointer to a double-precision floating-point variable that receives the execution rate. This rate is the number of times the operation function was called per second. **pdbMeasureRate()** attempts to compute a number of repetitions that results in a run time of about one second. (Calibration requires an additional second.) It's reasonably careful about timekeeping on systems with low-resolution clocks.

pdbMeasureRate() always returns PDB_NO_ERROR.

Example for pdbMeasureRate()

```
void SetupOpenGLState(void) {
        /* set all OpenGL state to desired values */
        }

void DrawTriangles(void) {
        glBegin(GL_TRIANGLE_STRIP);
                /* specify some vertices... */
        glEnd();
        }
main() {
                double rate;
                pdbOpen();
                if (pdbReadRate(NULL, "myApp", "triangles",
                   glGetString(GL_VERSION), &rate)
                      != PDB_NO_ERROR) {
                SetupOpenGLState();
                pdbMeasureRate(glFinish, DrawTriangles,
                    glFinish, 1, &rate);
                }
        printf("%g triangle calls per second\n", rate);
        pdbClose();
        }
```

Once a rate has been measured, it should be stored in the database by calling **pdbWriteRate()**, which has the following prototype:

`pdbStatusT pdbWriteRate (const char*` *machineName*`, const char*` *applicationName*`, const char*` *benchmarkName*`,const char*` *versionString*`, double` *rate*`)`

The first four arguments of **pdbWriteRate()** match the first four arguments of **pdbReadRate()**. The last argument is the performance measurement to be saved in the database.

pdbWriteRate() returns

- PDB_NO_ERROR if the performance measurement was added to the in-memory copy of the database
- PDB_OUT_OF_MEMORY if there was insufficient main memory to do so
- PDB_NOT_OPEN if the database is not open

When **pdbWriteRate()** is called, the in-memory copy of the performance database is marked "dirty." **pdbClose()** takes note of this and writes the database back to disk.

Example for pdbWriteRate()

```
main() {
        double rate;
        pdbOpen();
        if (pdbReadRate(NULL, "myApp", "triangles",
           glGetString(GL_VERSION), &rate)
              != PDB_NO_ERROR) {
                SetupOpenGL();
                pdbMeasureRate(glFinish, DrawTriangles,
                    glFinish, 1, &rate);
                pdbWriteRate(NULL, "myApp", "triangles",
                    glGetString(GL_VERSION), rate);
                }
        printf("%g triangle calls per second\n", rate);
        pdbClose();
        }
```

Using libisfast

The libisfast library is a set of demonstration routines that show how libpdb can be used to measure and maintain OpenGL performance data. libisfast is based on purely subjective performance criteria. If they're appropriate for your application, feel free to use them. If not, copy the source code and modify it accordingly.

In all cases that follow, the term "triangles" refers to a triangle strip with 37 vertices. The triangles are drawn with perspective projection, lighting, and smooth (Gouraud) shading. Unless otherwise stated, display-list-mode drawing is used. (This makes isfast yield more useful results when the target machine is being accessed over a network.)

The application must initialize isfast before performing any performance measurements, and clean up after the measurements are finished. On X11 systems these tasks are accomplished by calling

```
int IsFastXOpenDisplay(const char* displayName);
```

and

```
void IsFastXCloseDisplay(void);
```

respectively. **IsFastOpenXDisplay()** returns zero if the named display could not be opened, and nonzero if the display was opened successfully.

DepthBufferingIsFast() returns nonzero if depth buffered triangles can be drawn at least one-half as fast as triangles without depth buffering:

```
int DepthBufferingIsFast(void);
```

ImmediateModeIsFast() returns nonzero if immediate-mode triangles can be drawn at least one-half as fast as display-listed triangles:

```
int ImmediateModeIsFast(void);
```

Note that one significant use of **ImmediateModeIsFast()** may be to decide whether a "local" or a "remote" rendering strategy is appropriate. If immediate mode is fast, as on a local workstation, it may be best to use that mode and avoid the memory cost of duplicating the application's data structures in display lists. If immediate mode is slow, as is likely for a remote workstation, it may be best to use display lists for bulky geometry and textures.

StencillingIsFast() returns nonzero if stencilled triangles can be drawn at least one-half as fast as non-stencilled triangles:

```
int StencillingIsFast(void);
```

TextureMappingIsFast() returns nonzero if texture-mapped triangles can be drawn at least one-half as fast as non-texture-mapped triangles:

```
int TextureMappingIsFast(void);
```

Although the routines in libisfast are useful for a number of applications, you should study them and modify them for your own use. That way you'll explore the particular performance characteristics of your systems: their sensitivity to triangle size, triangle strip length, culling, stencil function, texture-map type, texture-coordinate generation method, and so on.

Index

D

E

F

G

M

P

Q

R

S

U

V

W

X

Z