OpenGL® on Silicon Graphics® Systems

OpenGL® on Silicon Graphics® Systems

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, "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, "Resource Control Extensions," discusses extensions that facilitate management of buffers and similar resources. Most of these extensions are GLX extensions.

Chapter 7, "Texturing Extensions,"explains how to use the texturing extensions, providing example code as appropriate.

Chapter 8, "Rendering Extensions," explains how to use extensions that allow you to customize the system's behavior during the rendering portion of the graphics pipeline. This includes blending extensions; the sprite, point parameters, reference plane, multisample, and shadow extensions; and the fog function and fog offset extensions.

Chapter 9, "Imaging Extensions," explains how to use extensions for color conversion (abgr, color table, color matrix), the convolution extension, the histogram/minmax extension, and the packed pixel extension.

Chapter 10, "Video Extensions," discusses extensions that can be used to enhance OpenGL video capabilities.

Chapter 11, "Miscellaneous OpenGL Extensions," explains how to use the instruments and list priority extensions as well as two extensions to GLU.

Chapter 12, "OpenGL Tools," explains how to use the OpenGL debugger (ogldebug) and discusses the glc OpenGL character renderer and (briefly) the gls OpenGL Streaming codec.

Chapter 13, "Tuning Graphics Applications: Fundamentals," starts with a list of general debugging hints. It then 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 14, "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 15, "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 16, "System−Specific Tuning,"provides information on tuning some specific Silicon Graphics systems: low−end systems, Indigo2 IMPACT systems, and RealityEngine systems. In this revision, it also includes information on O2 and InfiniteReality 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.

Appendix D, "Extensions on Different Silicon Graphics Systems," list all extensions currently supported on InfiniteReality, Impact, OCTANE, and O2 systems.

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 (and via the IRIS Insight viewer or the TechPubs library home page 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

Kilgard, Mark J. *OpenGL Programming for the X Window System*. Menlo Park, CA: Addison−Wesley Developer's Press. 1996. ISBN 0−201−48369−9.

Woo, Mason, Jackie Neider and Tom Davis. *OpenGL Programming Guide*: *The Official Guide to Learning OpenGL, Version 1.1*. Reading, MA: Addison Wesley Longman Inc. 1997. ISBN 0−201−46138−2.**(I)**

OpenGL Architecture Review Board; Renate Kempf and Chris Frazier, editors. *OpenGL Reference Manual*. *The Official Reference Document for OpenGL, Version 1.1*. Reading, MA: Addison Wesley Longman Inc. 1996. ISBN 0−201−46140−4.

OpenGL Porting Guide (007−1797−030)**(I)**

IRIS IM Programming Guide (007−1472−020)

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. 007−2479−001**(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.
Bold	Function names, with parentheses following the name—for example *glPolygonMode()*, arguments to command line options.

Note: Names of reference pages, such as glPolygonMode, are not functions. Reference page names appear in default font in hardcopy and in red text online. If you click the red text, the reference page will launch automatically.

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, because 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.

Changes in this Version of the Manual

This first revision of the manual contains the following changes:

Extensions removed. The manual has been updated for OpenGL 1.1. The functionality of some extensions was integrated into OpenGL 1.1 and the extensions have therefore been removed:

Texturing extensions	Texture objects, subtexture, copy texture.
Imaging extensions	Blend logic op
Miscellaneous extensions	Polygon offset, vertex array

Extensions added. The extension chapters have been reorganized as a finer−grained presentation. A number of extensions have been added:

Resource extensions	DMPbuffer extension
Texturing extensions	Texture filter4, filter4 parameters, texture LOD Bias, texture multibuffer, clipmap, texture select, texture add environment
Rendering extensions	Sprite, point parameters, reference plane, fog function, fog offset, shadow
Imaging extensions	Pixel texture
Video extensions	Swap barrier, swap group, video resize

Tools: The chapter discussing ogldebug, the OpenGL Debugger, has been updated to reflect ogldebug 1.1. In addition, a section on glc, the OpenGL character renderer, and gls, the OpenGL streaming utility, have been added to the chapter.

Performance: The performance chapters have been updated to include some additional information, most notably on InfiniteReality and O2 systems.

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"

"Extensions to OpenGL"

"Debugging and Performance Optimization"

"Location of Example Source Code"

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 Windows95. 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 run an application under different window systems, isolate the windowing code to minimize the number of files that must be special for each system.

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".

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 "GLX Extension to X". 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 discusses different tools and libraries for working with OpenGL in an X Window System environment. It starts with easy−to−use toolkits and libraries with less control and discusses the Xlib library\xd7 —which is more primitive but offers more control—last. 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 of the X Window System that are 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 is otherwise unavailable. Figure 1−1illustrates 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.

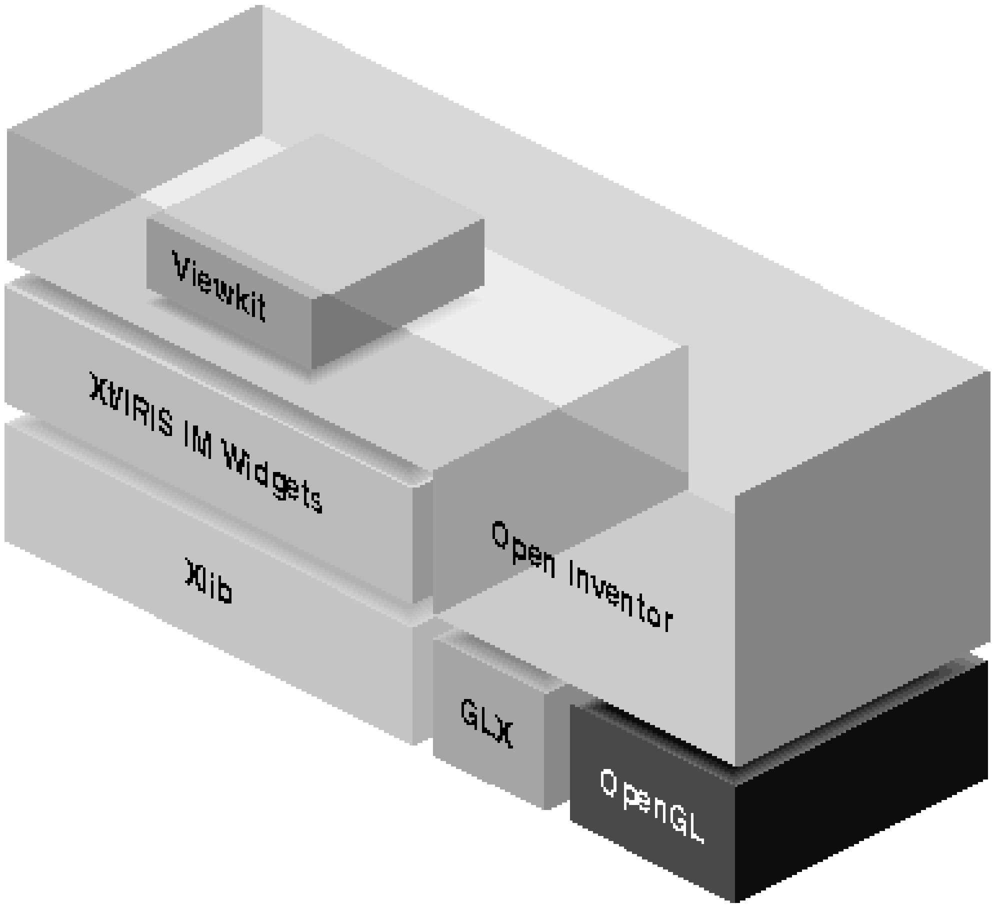


Figure 1−1 How X, OpenGL, and Toolkits Are Layered

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") 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 1−1because 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 widgets and components specific to Silicon Graphics. 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 are using libInventorXt, you need only link with libInventorXt (it automatically "exports" all of the routines in libInventor, so you never need to use **−lInventorXt−lInventor**; you need only **−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.

IRIS IM Widget Set

The IRIS IM widget set is an implementation of OSF/Motif provided by Silicon Graphics. You are 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" for further details.

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.

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 such as *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, because 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 7, "Texturing Extensions"
- Chapter 9, "Imaging Extensions"
- Chapter 11, "Miscellaneous OpenGL Extensions"
- Chapter 6, "Resource Control Extensions"

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 is 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 (Chapter 13 through Chapter 16) 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 13, "Tuning Graphics Applications: Fundamentals"
- Chapter 14, "Tuning the Pipeline"
- Chapter 15, "Tuning Graphics Applications: Examples"

In addition, there are tuning issues for particular hardware platforms. They are discussed in Chapter

16, "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
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"

"Libraries, Toolkits, and Tools"

"Integrating Your OpenGL Program With IRIS IM"

"Integrating OpenGL Programs With X—Summary"

"Compiling With OpenGL and Related Libraries"

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:

"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".

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.

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 SGI 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.

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 2−1provides 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	X11/xlib.h	−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

"GLX and Overloaded Visuals"

"GLX Drawables—Windows and Pixmaps"

"Rendering Contexts"

"Resources As Server Data"

"X Window Colormaps"

GLX and Overloaded 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 4−1for information about the visuals and their supported OpenGL rendering modes. The framebuffer configuration extension allows additional combinations. See "SGIX_fbconfig—The Framebuffer Configuration Extension".

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,

use *xdpyinfo* to find out about all the X visuals your system supports

use *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 xdpyinfo, glxinfo, and findvis reference pages for more information.

Not all X visuals support OpenGL rendering, but all X servers capable of OpenGL rendering have at least two OpenGL capable visuals. The exact number and type vary among different hardware systems. A Silicon Graphics system typically supports many more than the two required Open GL capable visuals. 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". Table 4−1illustrates which X visuals support which type of OpenGL rendering and whether the colormaps for those visuals are writable or not.

GLX Drawables—Windows and Pixmaps

As a rule, a drawable is something X can draw into, either a window or a pixmap (an exception is pbuffers, which are GLX drawables but cannot be used for X rendering). A GLX drawable is something both 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) Different ways of creating a GLX drawable are discussed in "Drawing−Area Widget Setup and Creation", "Creating a Colormap and a Window", and "Using Pixmaps".

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

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 don’t 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 As Server Data

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.

Because 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 "SGIX_fbconfig—The Framebuffer Configuration Extension".)

Note: 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.

X Window 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 their particular colormap is installed. The X window manager takes care of colormap installation and tries to make sure that the X client with input focus has its colormaps installed. On all systems, the colormap is a limited resource.

Every X window needs a colormap. If you are 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 assign one. For additional information, see "Using Colormaps". 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. A widget set 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 becomes 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 "Xlib Library".

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 2D rendering. 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 About Xt

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" for full bibliographic information and for pointers to additional documents about Motif and IRIS IM. The recently published book on OpenGL and X (Kilgard 1996) is particularly helpful for OpenGL developers.

Other Toolkits and Tools

Silicon Graphics makes several other tools and toolkits available that can greatly facilitate designing your IRIS IM interface. See "RapidApp", "Open Inventor", and "IRIS ViewKit" 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).

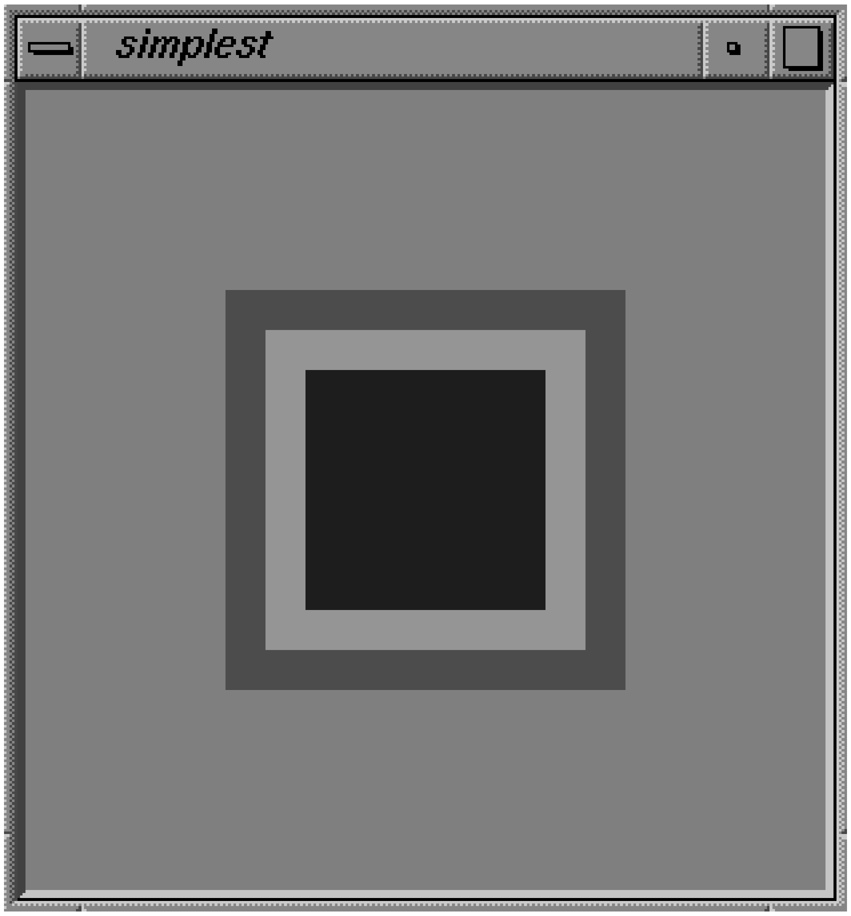


Figure 2−1 Display From simplest.c Example Program

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*/
 * calling glXWaitX makes sure that all x operations like *
 * XConfigureWindow to resize the window happen befor the *
 * OpenGL glViewport call.*/

static void
resize(Widget w, XtPointer client_data, XtPointer call) {
   GLwDrawingAreaCallbackStruct *call_data;
   call_data = (GLwDrawingAreaCallbackStruct *) call;
   glXWaitX();
   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" and "Creating a Colormap and a Window")

"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" and, for Xlib programming, "Xlib Event Handling".

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 are programming with Xlib, or *XtOpenApplication()* if you are working with widgets as in Example 2−1above. *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 glXQueryExtension 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 is 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, 6.2, and 6.3
GLX 1.2	IRIX 6.4

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 is 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()*. Typically, you specify the default screen that is returned by the *DefaultScreen()* macro.

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 2−1above, 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.

Note: Be aware that Xlib provides these three different but related visual data types. *glXChooseVisual()* actually returns an XVisualInfo*, which is a different entity form a visual* or a visual ID. *XCreateWindow()*, on the other hand, requires a visual*, not an XVisualInfo*.

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 have 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 "SGIX_fbconfig—The Framebuffer Configuration Extension".

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 have already opened.
vis	The visual you have 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 is not possible (for example, when rendering remotely). See "Direct and Indirect Rendering".

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"and "Creating a Colormap and a Window".

Binding the Context to the Window

If you are working with Xlib, bind the context to the window by calling *glXMakeCurrent()*.Example 3−2is a complete Xlib program and illustrates how the function is used.

If you are 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: "EXT_make_current_read—The Make Current Read Extension" allows you to attach separate read and write drawables to a GLX context.

Mapping the Window

A window can become visible only if it is 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 are 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 doesn't guarantee that the window is visible, only that it exists.

How you map the window on the screen depends on whether you have 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 XtRealizeWidget and XtCreateManagedChild reference pages.

Integrating OpenGL Programs With X—Summary

Table 2−2summarizes 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 glXQueryVers ion

"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 XCreateWind ow
"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

This sections lists link lines and the libraries that will be linked in.

−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.
−lm	Math library. Needed if your OpenGL program uses trigonometric or other special math routines.

Link Lines for Groups of Libraries

To use minimal OpenGL or additional libraries, use the following link lines:

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" illustrates how to create a window using IRIS IM drawing−area widgets and how to handle input and other events using callbacks.
- "Using Xlib" 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" looks at a simple example of using fonts with the *glXUseFont()* function.

Note: All integration aspects that are not dependent on your choice of Xlib or Motif are discussed in "Integrating Your OpenGL Program With IRIS IM" in Chapter 2, "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 that the OpenGL part of a program doesn't deal with. The section discusses the following topics:

- "About OpenGL Drawing−Area Widgets"
- "Drawing−Area Widget Setup and Creation"
- "Input Handling With Widgets and Xt"
- "Widget Troubleshooting"

About 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")

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 direct 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 is 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". 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). 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 cannot 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 is visible on screen. There is 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, uses a Form container widget and a Frame widget to draw the 3D box:

   ```
   toplevel = XtOpenApplication(&app, "simplest", NULL, 0, &argc, ar
   gv,
              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 have 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);
   ```

Note: Creating the widget doesn’t 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 user interface 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 cannot 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;
   glXWaitX();

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

Note: The X and OpenGL command streams are asynchronous, meaning that the order in which OpenGL and X commands complete is not strictly defined. In a few cases, it is important to explicitly synchronize X and OpenGL command completion. For example, if an X call is used to resize a window within a widget program, call *glXWaitX()* before calling *glViewport()* to ensure that the window resize operation is complete.

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 are using only one GLwMDrawingArea, you can call a routine to make the widget "current" just once, after initializing the widget. However, if you are 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").

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, because 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.

Note: If you create a GLwDrawingArea widget as a child of an already realized widget, it is 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" 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 are 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".

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:

```
*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 is 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 3−1is 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 are 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 Four), 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" and "Using Colormaps" 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 "General Tips for Debugging Graphics Programs".

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 doesn't), 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 doesn't 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 doesn't 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".

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 based on Xt, or if you are not 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 what Xlib routine caused the error.

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 that the OpenGL part of a program doesn't deal with. Because the complete example program in Chapter 2, "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 3−2lists 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 | KeyPressMa
```

```
sk;
    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);

    for(;;) 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 3−2calls *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 4−2for details on how to do this.

Note that you cannot 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". For information on overlays, which use a visual with a transparent pixel, see "Using Overlays".

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 have set. Here’s how Example 3−2does 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 doesn't 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 have created it.

```
void XSetWMProperties(Display *display, Window w,
                     XTextProperty *window_name, XTextProperty *icon_na
me,
                     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 generally 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. Popup overlay menus are an exception.

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 doesn't 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 is an example code fragment from *Xlib/mouse.c*, an Xlib program that uses mouse motion events. Example 3−3shows 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".

Note: 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 3−4prints 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"

"Using Overlays"

"Using Visuals"

"Using Colormaps"

"Stereo Rendering"

"Using Pixmaps"

"Performance Considerations for X and OpenGL"

"Portability"

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" 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" 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 cannot 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.

Note: 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 is 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". See "SGIX_fbconfig—The Framebuffer Configuration Extension" 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 is 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 is 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 is idle time while Xt is waiting for an event. If the function returns True, it is removed from the list of workprocs; if it returns False, it is kept on the list and called again when there is 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.glxwidge
t);
        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 is 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 togethe
r */
    XtAppContext appctx;
    Widget glxwidget;
    Boolean direct;
    XtWorkProcId animate_wpid;
} state;
time_animate:
static struct {        /* global UI variables; keep them togethe
r */
    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.glxwidg
et);
        }
        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.glxwidg
et);
        }
        break;
    case 1:
        exit(EXIT_SUCCESS);
        break;
    default:
        break;
    }
}
```

The *redraw_proc()* function has to register a new timeout each time it is called. Note that this differs from the workproc approach, where the application automatically continues animating as long as the system is not 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.glxwidg
    et);
    }
```

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 use them, discussing the following topics:

"Introduction to Overlays"

"Creating 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 is 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*.

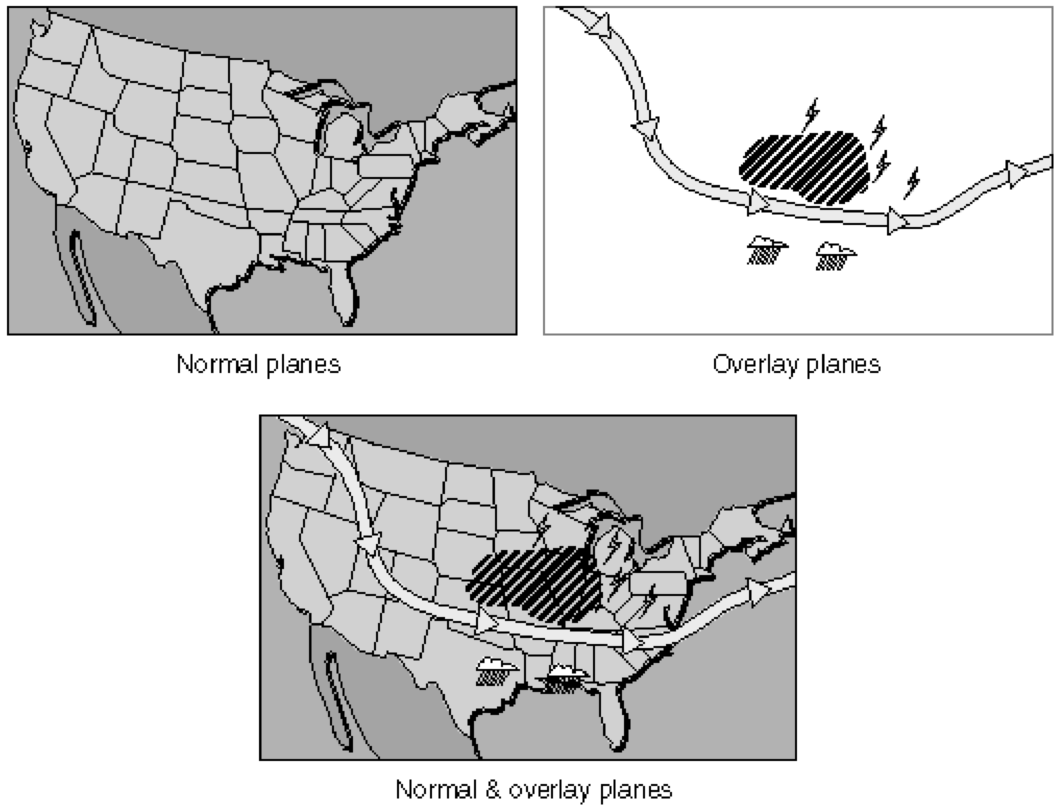


Figure 4−1 Overlay Plane Used for Transient Information

A special value in the overlay planes indicates transparency. On Silicon Graphics systems, it is 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 "Creating Overlays" 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.

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 cannot 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.

Creating 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 "EXT_visual_info—The Visual Info Extension" for more information.

Note: This example doesn't work if the visual info extension is not available (see "How to Check for OpenGL Extension Availability"). 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_E
XT,
                  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 (Indigo2 IMPACT), and high−end systems (RealityEngine) support 8−bit overlay planes.

Input events. The overlay window usually sits on top of the normal window. Thus, it receives 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 call *XRaiseWindow()* after the widget hierarchy has been realized.

Rubber Banding

Rubber banding can be used for cases where applications have to draw a few lines over a scene in response to a mouse movement. An example is the movable window outline that you see 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 is the best way to perform 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. Because 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" 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 4−1provides 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_m
enu,
                     &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, applicationShellWidge
tClass,
                  NULL, 0);
    dpy = XtDisplay(toplevel);

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

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

    glxwidget = XtVaCreateManagedWidget("glxwidget",
                   glwMDrawingAreaWidgetClass, frame, GLwNvisual
Info,
                   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"

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 4−1lists the X visuals support that support different types of OpenGL rendering, and tells you whether the colormaps for those visuals are writable or not. Visuals that are not 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[a]	yes
color index	PseudoColor	yes
color index	StaticColor*	no
not supported	GrayScale	yes
not supported	StaticGray	no

[a] 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 cannot 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, use *xdpyinfo* for a list of all visuals the server supports.

Use *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 are not 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".)

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 are 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 are 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 are 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"). 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). 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 doesn't 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 doesn't 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 doesn't reinstall the colormaps for your application until your application has the colormap focus again.

The *getColormap()* call defined in Example 4−2returns 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 "SGIX_fbconfig—The Framebuffer Configuration Extension".

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 because more bits must be updated per pixel, but this is not 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);

}
```

Note: 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"

"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 Indigo2 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" 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:

Because 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 are 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 on Stereo Rendering

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

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, Indigo2 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 Chapter 2 and Chapter 3: 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 Indigo2 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 a 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 "SGIX_pbuffer—The Pixel Buffer Extension".) A pixmap is an offscreen rendering area. On Silicon Graphics systems, pixmap rendering is not hardware accelerated.

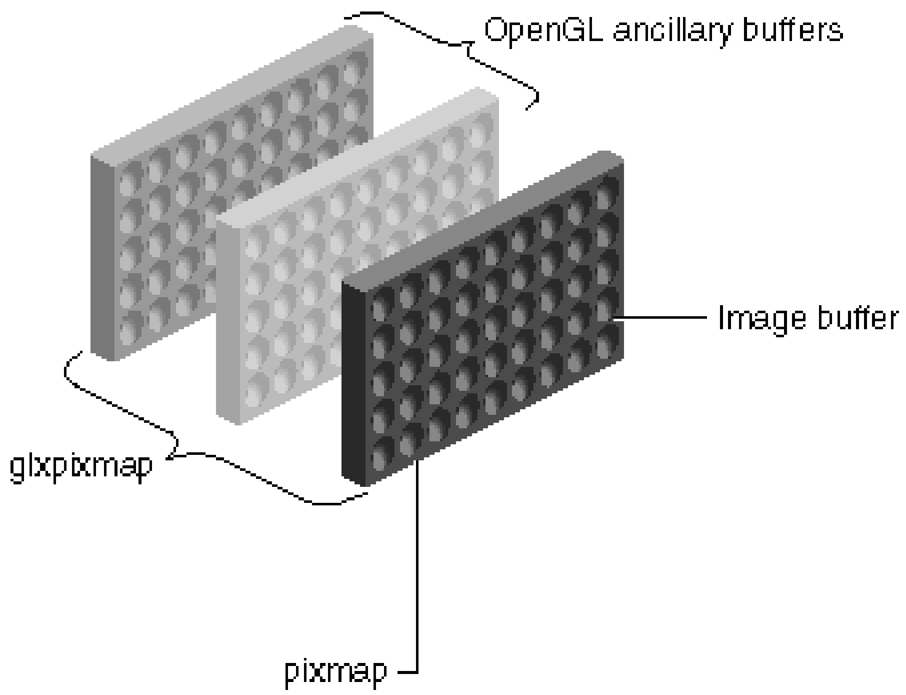


Figure 4−2 X Pixmaps and GLX Pixmaps

In contrast to windows, where drawing has no effect if the window is not visible, a pixmap can be drawn to at any time because 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.

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".)

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 are using pixmaps. If you ask for direct and your DISPLAY is remote, the library automatically switches to 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 doesn't 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 doesn't 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".

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 is 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 doesn't 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 fog function extension enhances the existing fog capability.

Several extensions provide functionality that existed in IRIS GL but is not available in OpenGL. 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"

"Finding Information About Extensions"

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 is 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.

SGI is used for extensions that are available 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 are 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 cannot compile or link a program that uses a certain extension if the client−side development environment doesn't 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 version 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.

Note that libdl interface allows users to dynamically load their own shared objects as needed. Applications can use this interface, particularly the *dlsym()* command, to compile their application on any system, even if some of the extensions used are not supported.

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 usually 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. Some extensions have been included in patch releases, so be careful when using this approach.

When an extension is incomplete, it is not advertised in the extensions string. Some of the RealityEngine extensions that were 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 6, "Resource Control Extensions," 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", taking these exceptions into account.

Finding Information About Extensions

You can find information about the extensions through reference pages, example programs, and extension specifications.

Reference Pages

For the most up−to−date information on extensions, see the following reference pages:

glintro	Information about the current state of extensions on your system.
glXintro	Information on GLX extensions.

Note that individual OpenGL reference pages have a MACHINE DEPENDENCIES section that lists the systems on which certain extension functions or options are implemented. Here is an example from the glSampleMaskSGIS reference page:

MACHINE DEPENDENCIES

Multisampling is supported only on RealityEngine, RealityEngine2, VTX and InfiniteReality systems. Currently it can be used with windows of Multisampling−capable Visual types, but not with pixmaps.

Example Programs

All complete example programs included in this guide (though not the short code fragments) are available in */usr/share/src/OpenGL* if you have the *ogl_dev.sw.samples* subsystem installed. You can also find example programs through the Silicon Graphics Developer Toolbox, http://www.sgi.com/Technology/toolbox.html.

Extension Specifications

Extension specifications describe extension functionality from the implementor's point of view. They are prepared to fit in with the OpenGL specification. Specification contain detailed information that goes beyond what developers usually need to know. If you need more detail on any of the extensions, search for its specification in the developer toolbox.

Chapter 6
Resource Control Extensions

This chapter discusses resource control extensions, which are 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*).

You can find information on GLX in several places:

Introductory information—See the glxintro reference page

In−depth coverage—See Appendix C, "OpenGL and Window Systems," of the *OpenGL Programming Guide* and *OpenGL Programming for the X Window System*

See "OpenGL and Associated Tools and Libraries" for bibliographical information).

This chapter explains how to use extensions to GLX. The extensions are presented in alphabetical order. You learn about

"EXT_import_context—The Import Context Extension"

"EXT_make_current_read—The Make Current Read Extension"

"EXT_visual_info—The Visual Info Extension"

"EXT_visual_rating—The Visual Rating Extension"

The following sections describe extensions that are experimental:

"SGIX_dm_pbuffer—The Digital Media Pbuffer Extension"

"SGIX_fbconfig—The Framebuffer Configuration Extension"

"SGIX_pbuffer—The Pixel Buffer Extension"

Note: Using OpenGL in an X Window System environment is discussed in the following chapters of this guide:

Chapter 2, "OpenGL and X: Getting Started"

Chapter 3, "OpenGL and X: Examples"

Chapter 4, "OpenGL and X: Advanced Topics"

EXT_import_context—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 work effectively with this extension, you must first understand direct and indirect rendering. See "Direct and Indirect Rendering" 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 *glXGetContextIDEXT()*:

```
GLXContextID glXGetContextIDEXT(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, *glXGetContextIDEXT()* does not flush any pending events.

To create a GLX context, given the XID of an existing GLX context, call *glXImportContextEXT()*. You can use this function in place of *glXCreateContext()* to share another process' indirect rendering context:

```
GLXContext glXImportContextEXT( 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, *glXImportContextEXT()* must allocate memory to store client−side information.

A call to *glXImportContextEXT()* doesn't 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*glXImportContextEXT()*, call *glXFreeContextEXT()*:

```
void glXFreeContextEXT(Display *dpy, GLXContext ctx)
```

glXFreeContextEXT() doesn't 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()*, which has the following prototype:

```
Display * glXGetCurrentDisplayEXT( void );
```

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()* doesn't flush any pending events.

To obtain the value of a context's attribute, call *glXQueryContextInfoEXT()*:

```
int glXQueryContextInfoEXT( Display *dpy, GLXContext ctx,
                           int attribute,int *value )
```

The values and types corresponding to each GLX context attribute are listed in Table 6−1.

Table 6−1 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, glXGetContextIDEXT, glXImportContextEXT, glXFreeContextEXT, glXQueryContextInfoEXT

EXT_make_current_read—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()*, which has the following prototype:

```
Bool glXMakeCurrentReadSGI( Display *dpy,GLXDrawable draw,
                            GLXDrawable read, GLXContext gc )
```

where

dpy	Specifies the connection to the X server.
draw	A GLX drawable that receives the results of OpenGL drawing operations.
read	A GLX drawable that provides pixels for *glReadPixels()* and *glCopyPixels()* operations.
gc	A GLX rendering context to be attached to draw and read.

Read and Write Drawables

In GLX 1.1, you associate a GLX context with one drawable (window or pixmap) by 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.

In effect, the following calls are equivalent:

```
MakeCurent(context, win)
MakeCurrentRead(context, win, win)
```

Having both a read and a write drawable is useful, 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()*, *glCopyTexImage()*, or *glCopyTexSubImage()*. It is also use by any OpenGL extension that sources images from the framebuffer in the manner of *glReadPixels()*, *glCopyPixels()*, *glCopyTexImage()*, or *glCopyTexSubImage()*.

Here is some additional information about the two drawables:

The two drawables do not need to have the same ancillary buffers (depth buffer, stencil buffer, and so on).

The read drawable does not 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 does not 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.

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 be the same. If it is not, *glXMakeCurrentReadSGI()* generates a BadMatch X protocol error.

Retrieving the Current Drawable's Name

glXGetCurrentReadDrawableSGI() returns the name of the GLXDrawable currently being used as a pixel query source.

If *glXMakeCurrent()* specified the current rendering context, then *glXGetCurrentReadDrawableSGI()* returns the drawable specified as *draw* by that glXMakeCurrent call.

If *glXMakeCurrentReadSGI()* specified the current rendering context, then *glXGetCurrentReadDrawableSGI()* returns the drawable specified as *read* by that *glXMakeCurrentReadSGI()* call.

If there is no current read drawable, *glXGetCurrentReadDrawableSGI()* returns None.

New Functions

glXefReadSGI.

EXT_visual_info—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 )
```

The following heuristics determine which visual is chosen:

Table 6−2 Heuristics for Visual Selection

If...	And GLX_X_VISUAL_TYPE_EXT is...	The result is...
GLX_RGBA is in *attrib_list*	GLX_TRUE_COLOR_EXT	TrueColor visual
	GLX_DIRECT_COLOR_EXT	DirectColor visual
	GLX_PSEUDO_COLOR_EXT, GLX_STATIC_COLOR_EXT, GLX_GRAY_SCALE_EXT, or GLX_STATIC_GRAY_EXT	Visual Selection fails
	Not in *attrib_list*, and if all other attributes are equivalent...	A TrueColor visual (GLX_TRUE_COLOR_EXT) is chosen in preference to a DirectColor visual (GLX_DIRECT_COLOR_EXT)
GLX_RGBA is not in *attrib_list*	GLX_PSEUDO_COLOR_EXT	PseudoColor visual
	GLX_STATIC_COLOR_EXT	StaticColor visual
	GLX_TRUE_COLOR_EXT, GLX_DIRECT_COLOR_EXT, GLX_GRAY_SCALE_EXT, or GLX_STATIC_GRAY_EXT	Visual selection fails
	Not in *attrib_list* and if all other attributes are equivalent...	A PseudoColor visual (GLX_PSEUDO_COLOR_EXT) is chosen in preference to a StaticColor visual (GLX_STATIC_COLOR_EXT)

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 from *glXChooseVisual()*, use *XFree()*.

Note that GLX_VISUAL_TYPE_EXT can also be used with *glXGetConfig()*.

Using Transparent Pixels

How you specify that you want a visual with transparent pixels depends on the existing attributes:

If...	Then call *glXChooseVisual()* and specify as the value of GLX_TRANSPARENT_TYPE_EXT...
GLX_RGBA is in *attrib_list*	GLX_TRANSPARENT_RGB_EXT

GLX_RGBA is not in *attrib_list*	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.

"Creating Overlays" presents an example program that uses a transparent visual for the overlay window.

EXT_visual_rating—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.

SGIX_dm_pbuffer—The Digital Media Pbuffer Extension

The Digital Media Pbuffer extension, SGIX_dm_pbuffer, introduces a new type of GLXPbuffer, the DMbuffer. Images generated by digital media libraries in DMbuffer form can be used directly by OpenGL as renderable buffers or as the pixel source for texture images.

Note: Note:

The SGIX_dm_pbuffer extension is currently supported only on O2 systems. This discussion therefore focuses on the buffer configurations available on O2 systems.

This section explains how to use the Digital Media Pbuffer extension in the following sections:

"Creating a Digital Media Pbuffer" provides a conceptual introduction to the steps involved in using the extension.

"Compatibility Conditions" discusses image layout and pixel formats in for the different libraries. This background information is used when an application creates DMbuffers that are compatible with DMPbuffers.

"OpenGL Rendering to DMbuffers" provides an example program that illustrates the material discussed in the other two sections.

"DMbuffers as OpenGL Textures" explains conditions under which DMbuffers can be used as OpenGL textures. It also includes an example code fragment.

Creating a Digital Media Pbuffer

Creating a digital media Pbuffer involves three separate conceptual steps, explained in the following sections. "OpenGL Rendering to DMbuffers" further illustrates each step in the context of an example program.

Table 6−3 Steps for Creating a Digital Media Pbuffer

Step...	Discussed in...
1	"Creating a DMBuffer"
2	"Creating a Digital Media Pbuffer"
3	"Associating Pbuffer and DMbuffer"

Creating a DMBuffer

DMbuffers are a class of buffer common to video, JPEG decompression and other digital media libraries. They permit the sharing and exchange of images in various formats. A graphical DMbuffer is essentially a chunk of memory used to store a single image or, in the special case of mipmapped DMbuffers, a set of images.

To use the dm_pbuffer extension, you have to create a DMbufferPool with characteristics that match the Pbuffer you want to associate with the DMbuffer. Follow these conceptual steps (in an actual program, memory allocation and other issues are also part of the process):

1. Call *dmBufferSetPoolDefaults()* to specify the parameters of the DMbuffers you want to create.

   The DMparams identify the DMbuffer when it is passed to OpenGL. (see Example 6−1). The following elements in the structure must be compatible with the characteristics of the Pbuffer:

   DM_IMAGE_WIDTH and DM_IMAGE_HEIGHT

   DM_IMAGE_PACKING—(see "Pixel Formats")

   DM_IMAGE_LAYOUT—(see "Compatibility Conditions")

2. Call *dmBufferCreatePool()* to create a DMbufferPool.

   All the buffers in the pool will have the characteristics specified in step 1.

3. Once the buffer pool is created, DMbuffers are obtained with a call to either *dmBufferAllocate()*, *vlEventToDMBuffer()*, or *dmICReceive()*, depending on the application generating DMbuffers for OpenGL.

Creating a Digital Media Pbuffer

A pixel buffer, or Pbuffer, is a window−independent, non−visible rendering buffer for an OpenGL renderer. Pbuffers are supported by the Pbuffer extension; see "SGIX_pbuffer—The Pixel Buffer Extension". A digital media Pbuffer is a special kind of Pbuffer.

To create a digital media Pbuffer, an application calls *glXCreateGLXPbufferSGIX()*, specifying the GLX_DIGITAL_MEDIA_PBUFFER_SGIX attribute.

The resulting Pbuffer is identical in all respects to a standard Pbuffer except that its primary color buffer does not exist until the Pbuffer is associated with a compatible DMbuffer for the first time. All other buffers (depth, stencil, accumulation) defined by the FBConfig for the pbuffer are allocated by OpenGL.

Associating Pbuffer and DMbuffer

To associate a Pbuffer with a compatible DMbuffer, applications call *glXAssociateDMPbufferSGIX()* which has the following prototype:

```
Bool glXAssociateDMPbufferSGIX( Display *dpy,GLXPbufferSGIX pbuffer,
                                DMparams *params,DMbuffer dmbuffer )
```

where

dpy	Connection to an X server.
pbuffer	GLX pixel buffer target of the associate operation.
params	Parameter list that describes the format of the images in the DMbuffer that is to be associated with the pixel buffer.
dmbuffer	DMbuffer to be used as the front left color buffer.

The call to *glXAssociateDMPbufferSGIX()* must be issued before the pbuffer can be made current for the first time, as either a read or write drawable. Once associated with a pbuffer, all rendering to, or read and copy operations from the pbuffer's color buffer will access the DMbuffer directly.

Compatible DMbuffers can be associated in sequence with the same pbuffer while the pbuffer is current. A DMbuffer remains associated either until it is replaced by another associate command, or until the pbuffer is destroyed. Once the DMbuffer is released, it is freed only if it has no remaining clients on the system. DMbuffers are local resources, and a DMPbuffer can be current only to a direct GLXContext.

Compatibility Conditions

A pbuffer and DMbuffer can be associated only if their image layout and pixel formats are compatible. This section provides some background information on these two topics.

Image Layouts

When an application creates a pool of DMbuffers, it has to choose between two types of DMbuffer image layout, linear and graphics, specified with the DM_IMAGE_LAYOUT parameter in the DMparams structure. Table 6−4lists OpenGL commands that are compatible with DMbuffers of each layout.

Table 6−4 Linear and Graphics Layout

	Linear Layout	Graphics Layout
VL Layout	VL_LAYOUT_ LINEAR	VL_LAYOUT_ GRAPHICS, VL_LAYOUT_ MIPMAP
DM image Layout	DM_IMAGE_LAYOUT_ LINEAR	DM_IMAGE_LAYOUT_ GRAPHICS, DM_IMAGE_LAYOUT_ MIPMAP
OpenGL commands	glDrawPixels, glReadPixels, glTexImage2D	glXAssociateDMPbufferSGIX, glCopyTexSubImage2D

Only DMbuffers with a graphics layout can be associated with a DMPbuffer. These DMbuffers cannot be mapped, and so can be accessed only through digital media or graphics library commands, not directly by the application.

DMbuffers with linear image layout can be mapped, and can be passed by address as the pixels parameter to *glDrawPixels()*, *glReadPixels()* and *glTexImage2D()*.

Pixel Formats

There are three internal pixel formats that are shared by the video, digital media and graphics libraries. The video library (libvl), the digital media library (libdmedia), and the graphics library (libGL) each have different designations for the same internal format, as illustrated in Table 6−5.

Table 6−5 Pixel and Texel Formats (Video, Digital Media and Graphics)

	libvl	libmedia	libGL− texel, pixel
rgba−8888	VL_PACKING_ABGR8	DM_IMAGE_PACKING_ RGBA	GL_RGBA8_EXT, GL_RGBA with GL_UNSIGNED_BYTE
rgba−5551	VL_PACKING_ARGB_1555	DM_IMAGE_PACKING_ XRGB5551	GL_RGB5_A1_EXT, GL_RGBA with GL_UNSIGNED_BYTE_5_ 5_5_1_EXT
rgb−332	VL_PACKING_X444_332	DM_IMAGE_PACKING_ RGB332	(332 texel not supported) GL_RGB with GL_UNSIGNED_BYTE_3_ 3_2_EXT

The DM_IMAGE_PACKING parameter of the DMparams structure should be set to a format that matches the component depths described by the DMPbuffer FBConfig. Video applications also need to initialize the path to a matching video library format.

OpenGL Rendering to DMbuffers

Setup required for rendering to a DMPbuffer involves three basic steps, illustrated by example code fragments in the following section:

1. "Creating DMParams Structure and DMBuffer Pool"

2. "Creating a Compatible DMPbuffer"

3. "Associating the DMBuffer With the DMPbuffer"

Creating DMParams Structure and DMBuffer Pool

The following sample code fragment creates a DMparams structure, and a pool of DMbuffers that are suitable for use by video and GL. The buffers are 640 x 480 with a graphics layout and 32−bit RGBA format.

Example 6−1 Creating a DMparams Structure and DMbuffer Pool

```
DMparams *imageParams, *poolParams;
DMbufferpool bufferPool;
DMpacking dmPacking = DM_IMAGE_PACKING_RGBA;
DMimagelayout dmLayout = DM_IMAGE_LAYOUT_GRAPHICS;
DMboolean cacheable = DM_FALSE;
DMboolean mapped = DM_FALSE;
int bufferCount = NUMBER_OF_BUFFERS_NEEDED_BY_APPLICATION;
DMbuffer buffer[NUMBER_OF_BUFFERS_NEEDED_BY_APPLICATION];
DMstatus s;

/* Create and initialize image params. */
s = dmParamsCreate( &imageParams );
s = dmSetImageDefaults( imageParams, 640, 480, dmPacking );
s = dmParamsSetEnum( imageParams, DM_IMAGE_LAYOUT, dmLayout );

/* Set up a VL video path before creating the DMbuffer pool. */

/* Create and initialize pool params using VL & GL dm utilities. */
s = dmParamsCreate( &poolParams );
s = dmBufferSetPoolDefaults( count, 0, cacheable, mapped );

s = vlDMPoolGetParams( vlServer, vlPath, vlNode, poolParams );

s = dmBufferGetGLPoolParams( imageParams, poolParams );

/* Set buffer count and create pool. */
bufferCount += dmParamsGetInt( poolParams, DM_BUFFER_COUNT );
dmParamsSetInt( poolParams, DM_BUFFER_COUNT, bufferCount );

s = dmBufferCreatePool( poolParams, &bufferPool );
dmParamsDestroy( poolParams );
```

Creating a Compatible DMPbuffer

The next step is to create a DMPbuffer with the same size and format as the DMbuffers that are to be rendered to.

Example 6−2 Creating a Digital Media Pbuffer

```
GLXFBConfigSGIX *config;
GLXPbufferSGIX pbuffer;
GLXContext context;
int configAttribs [] = {
            GLX_DOUBLEBUFFER, True,
            GLX_RED_SIZE, 8,
            GLX_GREEN_SIZE, 8,
            GLX_BLUE_SIZE, 8,
            GLX_ALPHA_SIZE, 8,
            GLX_DRAWABLE_TYPE_SGIX, GLX_PBUFFER_BIT_SGIX,
            (int) None };
int pbufAttribs [] = {
            GLX_DIGITAL_MEDIA_PBUFFER_SGIX, True,
            (int) None };

config = glXChooseFBConfigSGIX(display, screen, configAttribs);

pbuffer = glXCreateGLXPbufferSGIX(display, *config, 640, 480,
                           pbufAttribs);
context = glXCreateContextWithConfigSGIX(display, *config,
                        GLX_RGBA_TYPE_SGIX, NULL, True);
```

Associating the DMBuffer With the DMPbuffer

Finally the DMbuffer is allocated and associated with the DMPbuffer and made current to a context. Applications typically cycle through a sequence of DMbuffers, rendering to them, or copying them to OpenGL textures. Freeing the DMbuffer after it has been associated allows the buffer to return to the pool for reuse once it is released by the OpenGL pbuffer or texture object.

Example 6−3 Associating a DMbuffer With a DMPbuffer

```
DMparams *imageParams = ...;
DMbufferpool bufferPool = ...;
DMbuffer dmBuffer;
DMstatus s;

/* associate the first DMbuffer before making current */
s = dmBufferAllocate( bufferPool, &dmBuffer );
glXAssociateDMPbufferSGIX( display, pbuffer, imageParams, dmBuffer);
glXMakeCurrent( display, pbuffer, context );

for(i = 0; i < bufferCount; i++)i {

    /* perform GL rendering operations to the DMbuffer */

    dmBufferFree( dmBuffer );
    s = dmBufferAllocate( bufferPool, &dmBuffer );
    glXAssociateDMPbufferSGIX(display, pbuffer, imageParams, dmBuffe
r);
}
```

DMbuffers as OpenGL Textures

Under certain conditions, the SGIX_dm_pbuffer implementation on O2 permits the direct use of a

DMbuffer as a GL texture. The benefits are optimized texture loading for DMbuffers generated as video and JPEG images, and for textures rendered as images to a DMPbuffer.

After DMBuffer and pbuffer have been associated, applications can call *glXMakeCurrentReadSGI()* with a DMPbuffer as the read drawable, then call *glCopyPixels()* or *glCopyTexImage2D()* to copy the contents of the associated DMbuffer to another drawable or to a texture. These copy operations behave as they would with any standard read drawable.

The following conditions allow for a "copy by reference" of the currently associated DMbuffer to a texture object (also see the reference page for glCopyTexSubImage2D):

glCopyTexSubImage2D() is used to copy the entire texture image from the DMPbuffer.

The DMPbuffer and target 2D texture object match in terms of width, height, and depth of RGBA components. See "Pixel Formats" for comparable formats.

The texture object is 64 (or more) texels in its largest dimension.

If the DMbuffer image layout is DM_IMAGE_LAYOUT_MIPMAP, then the GL_GENERATE_MIPMAP_SGIS texture parameter must also be set to TRUE for the texture object at the time of the copy.

Only the default pixel transfer operations are enabled at the time of the copy.

After a DMbuffer is copied by reference to the texture object it remains associated as the texture, even once the association to the source DMPbuffer changes, and until the texture object is destroyed or the texture image is updated through another OpenGL command.

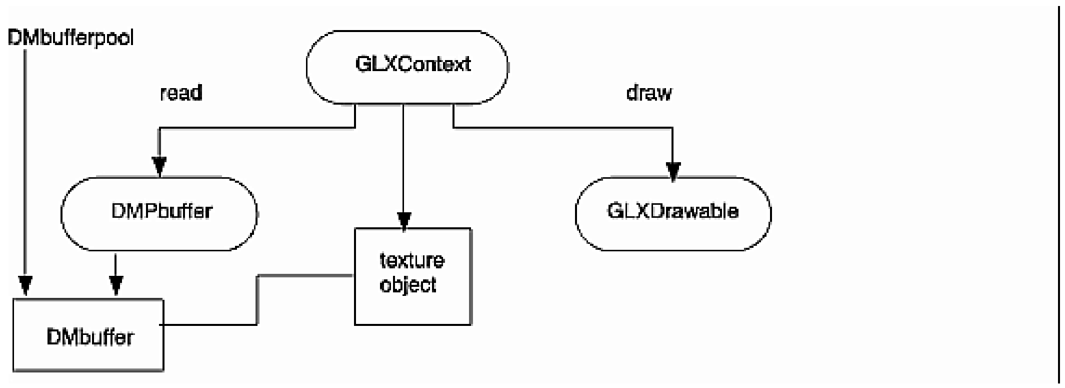


Figure 6−1DMPbuffers and DMbuffers

The following example demonstrates the optimized case for copying a DMbuffer by reference to a texture object. The source DMPbuffer and DMbuffer differ from previous examples only in size; they are 512 square to allow for direct use as a OpenGL texture.

Example 6−4 Copying a DMbuffer to a Texture Object.

```
/* Make DMPbuffer current as a read drawable */
pbuffer = glXCreateGLXPbufferSGIX(display, *config, 512, 512, attrib
s);
glXAssociateDMPbufferSGIX( display, pbuffer, imageParams, dmBuffer);
glXMakeCurrentReadSGI( display, drawable, pbuffer, context );
```

```
/* Create and init a compatible GL texture object with NULL image */

glGenTextures(1, &texObj);
glBindTexture(GL_TEXTURE_2D, texObj);
glTexImage2D(GL_TEXTURE_2D, level = 0, GL_RGBA8, w = 512, h = 512,
            GL_RGBA, GL_UNSIGNED_BYTE, NULL);

/* copy the DMbuffer by reference to the texture object */

glCopyTexSubImage(GL_TEXTURE_2D, level = 0, xoff = 0, yoff = 0,
                  x = 0, y = 0, w = 512, h = 512);
```

New Function

glXAssociateDMPbufferSGIX

SGIX_fbconfig—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 GLXFBConfigSGIX construct (also called FBConfig).

It relaxes the "similarity" requirement when associating a current context with a drawable.

It supports RGBA rendering to one− and two−component windows (luminance and luminance alpha rendering) and GLX pixmaps as well as pbuffers (pixel buffers). Pbuffers are discussed in "SGIX_pbuffer—The Pixel Buffer Extension".

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 "SGIX_pbuffer—The Pixel Buffer Extension")

if you want to render luminance data to a TrueColor visual

instead of *glXChooseVisual()*, because 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 the extension provides.

Describing a Drawable With a GLXFBConfigSGIX 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 GLXFBConfigSGIX, 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 FBConfig.

A particular FBConfig is not required to work with all GLX drawables. For example, it is possible for implementations to export FBConfigs that work only with GLX pixmaps.

Less−Rigid Similarity Requirements When Matching Context and Drawable

In OpenGL without the extension, if 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 FBConfig that the drawable was created with. For example, if the context was created for RGBA rendering, it can be used only if the FBConfig supports RGBA rendering.

All color buffers and ancillary buffers that exist in both FBConfigs 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 an FBConfig that has only a front left buffer with red, green, and blue sizes of 8. However, it is compatible with an FBConfig 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 framebuffer values and the green and blue fragments are discarded.

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 that it allows you to work with single−component images with texture mapping, then use a pseudo−color visual to map the luminance values to color.

GLXFBConfigSGIX Constructs

A GLXFBConfigSGIX (FBConfig) describes the format, type, and size of the color and ancillary buffers for a GLX drawable. If the GLX drawable is a window, then the FBConfig 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 FBConfig.

Choosing a GLXFBConfigSGIX Construct

Use *glXChooseFBConfigSGIX()* to get GLXFBConfigSGIX constructs that match a list of attributes or to get the list of GLXFBConfigSGIX constructs (FBConfigs) that are available on the specified screen.

```
GLXFBConfigSGIX *glXChooseFBConfigSGIX(Display *dpy, int screen,
                                        const int *attrib_list, int *nitems)
```

If *attrib_list* is NULL, *glXChooseFBConfigSGIX()* returns an array of FBConfigs that are available on the specified screen; otherwise this call returns an array of FBConfigs that match the specified attributes. Table 6−6shows only attributes added by this extension; additional attributes are listed on the glXChooseVisual reference page.

Table 6−6 Visual Attributes Introduced by the FBConfig Extension

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 FBConfig.

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	FBConfigs 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 FBConfigs 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 FBConfigs 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 that "don’t care" means that the default behavior is to have no preference when searching for a matching FBConfig.

Table 6−7illustrates how each attribute is matched.

Table 6−7 FBConfig 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
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 FBConfigs on the screen (*attrib_list* is NULL).

Retrieve an FBConfig with a given ID specified with GLX_FBCONFIG_ID_SGIX.

Retrieve the FBConfig that is the best match for a given list of visual attributes.

Retrieve first a list of FBConfigs that match some criteria, for example, each FBConfig 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 FBConfig is obtained, you can use it to create a GLX pixmap, window, or pbuffer (see "SGIX_pbuffer—The Pixel Buffer Extension"). 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 matching FBConfig exists, or if an error occurs (that is, an undefined GLX attribute is encountered in *attrib_list*, *screen* is invalid, or *dpy* doesn’t support the GLX extension) then NULL is returned.

If *attrib_list* is not NULL and more than one FBConfig is found, then an ordered list is returned with the FBConfigs that form the "best" match at the beginning of the list. ("How an FBConfig

Is Selected" 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 FBConfigs 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 FBConfigs 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 FBConfigs that support both windows and GLX pixmaps. The default value for GLX_DRAWABLE_TYPE_SGIX is GLX_WINDOW_BIT_SGIX.

If an FBConfig 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 doesn't 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 FBConfig. This attribute is always true if the FBConfig supports windows and/or GLX pixmaps.

Retrieving FBConfig Attribute Values

To get the value of a GLX attribute for an FBConfig, call

```
int glXGetFBConfigAttribSGIX(Display *dpy, GLXFBConfigSGIX 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: An FBConfig 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,
                                          GLXFBConfigSGIX config)
```

If *config* is a valid FBConfig 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 an FBConfig, given visual information:

```
GLXFBConfigSGIX glXGetFBConfigFromVisualSGIX(Display *dpy, XVisualIn
fo *vis)
```

If the visual is valid and supports OpenGL rendering (that is, if the GLX visual attribute GLX_USE_GL is GL_TRUE) then the associated FBConfig is returned; otherwise NULL is returned.

To create a GLX rendering context or a GLX pixmap using an FBConfig, call *glXCreateContextWithConfigSGIX()* or *glXCreateGLXPixmapWithConfigSGIX()*, which have the following prototypes:

```
GLXContext glXCreateContextWithConfigSGIX( Display *dpy,
                                          GLXFBConfigSGIX config,
                                          int render_type,
                                          GLXContext share_list,
                                          Bool direct )
GLXPixmap glXCreateGLXPixmapWithConfigSGIX( Display *dpy,
                                            GLXFBConfigSGIX config,
                                            Pixmap pixmap )
```

The functions are similar to *glXCreateContext()* and *glXCreateGLXPixmap()*. See the glXCreateContextWithConfigSGIX and glXCreateGLXPixmapWithConfigSGIX reference pages for detailed information.

How an FBConfig Is Selected

If more than one FBConfig matches the specification, they are prioritized as follows (Table 6−7 summarizes this information):

Preference is given to FBConfigs 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 FBConfigs that have GLX_ALPHA_SIZE set to zero; otherwise preference is given to FBConfigs that have the largest GLX_ALPHA_SIZE value.

If the requested number of GLX_AUX_BUFFERS is zero, preference is given to FBConfigs that have GLX_AUX_BUFFERS set to zero; otherwise preference is given to FBConfigs 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 FBConfigs that also have that size set to zero; otherwise preference is given to FBConfigs 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 FBConfigs that also have these attributes set to zero; otherwise preference is given to FBConfigs that have the smallest size.

If GLX_X_VISUAL_TYPE_EXT is not specified but there is an X visual associated with the FBConfig, the visual type is used to prioritize the FBConfig.

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 "EXT_visual_rating—The Visual Rating Extension" for more information.

New Functions

glXGetFBConfigAttribSGIX, glXChooseFBConfigSGIX, glXCreateGLXPixmapWithConfigSGIX, glXCreateContextWithConfigSGIX, glXGetVisualFromFBConfigSGIX, glXGetFBConfigFromVisualSGIX.

SGIX_pbuffer—The Pixel Buffer Extension

You can use the pixel buffer extension, SGIX_pbuffer, to define a pixel buffer (GLXPbuffer or pbuffer for short).

Note: This extension is an SGIX (experimental) extension. The interface or other aspects of the extension may change.

About GLXPbuffers

A GLXPbuffer is an additional non−visible rendering buffer for an OpenGL renderer. It has the following distinguishing characteristics:

Support hardware−accelerated rendering. Pbuffers support hardware−accelerated rendering in an off−screen buffer, unlike pixmaps, which typically do not allow accelerated rendering.

Window independent. Pbuffers differ from auxiliary buffers (aux buffers) because they are not related to any displayable window, so a pbuffer may not be the same size as the application's window, while an aux buffer must be the same size as its associated window.

PBuffers and Pixmaps

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 be described only with an FBConfig; 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.

Volatile and Preserved Pbuffers

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.

Creating a PBuffer

To create a GLXPbuffer, call *glXCreateGLXPbufferSGIX()*:

```
GLXPbufferSGIX glXCreateGLXPbufferSGIX(Display *dpy, GLXFBConfigSGIX
 config,
                 unsigned int *width, unsigned int *height, int attrib_lis
t)
```

This call 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, for example, when the application windows are iconified.

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 FBConfig then a GLXBadFBConfigSGIX error is generated; if *config* doesn't support pbuffers, a BadMatch X protocol error is generated.

Rendering to a GLXPbuffer

Any GLX rendering context created with an FBConfig or X visual that is compatible with an FBConfig may be used to render into the pbuffer. For the definition of "compatible," see the reference pages for 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, because grabbing the X server locks out other X clients, you should do it only for short periods of time.) Clients that don't wish to grab the X server can check whether the data returned by *glReadPixels()* is valid by calling *XSync()* and then checking the event queue for "buffer clobber" events (assuming that any previous clobber events were pulled off of the queue before the *glReadPixels()* call).

To destroy a GLXPbuffer call *glXDestroyGLXPbufferSGIX()*:

```
void glXDestroyGLXPbufferSGIX(Display *dpy, GLXPbufferSGIX pbuf)
```

To query an attribute associated with a GLXPbuffer, call *glXQueryGLXPbufferSGIX()*:

```
void glXQueryGLXPbufferSGIX(Display *dpy, GLXPbufferSGIX pbuf, int attr
ibute
                           unsigned int *value)
```

To get the FBConfig for a GLXPbuffer, first retrieve the ID for the FBConfig and then call *glXChooseFBConfigSGIX()*. See "SGIX_fbconfig—The Framebuffer Configuration Extension".

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 affecte
d*/
  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.

Events are sent to the application and queried using the normal X even commands (*XNextEvent()*, *XPending()*, and so on). The *mask* value returned in the event structure indicates which color and ancillary buffers were affected. The following values can be set in the event structure:

```
GLX_FRONT_LEFT_BUFFER_BIT_SGIX
GLX_FRONT_RIGHT_BUFFER_BIT_SGIX
GLX_BACK_LEFT_BUFFER_BIT_SGIX
GLX_BACK_RIGHT_BUFFER_BIT_SGIX
GLX_AUX_BUFFERS_BIT_SGIX
GLX_DEPTH_BUFFER_BIT_SGIX
GLX_STENCIL_BUFFER_BIT_SGIX
GLX_ACCUM_BUFFER_BIT_SGIX
GLX_SAMPLE_BUFFERS_BIT_SGIX
```

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 vary, depending 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.

Calling *glXSelectEventSGIX()* overrides any previous event mask that was set by the client for the drawable. Note that it doesn't affect the event masks that other clients may have specified for a drawable, because 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.

Chapter 7
Texturing Extensions

This chapter explains how to use the different OpenGL texturing extensions. The extensions are discussed in alphabetical order, by extension name:

"EXT_texture3D—The 3D Texture Extension"

"SGI_texture_color_table—The Texture Color Table Extension"

"SGIS_detail_texture—The Detail Texture Extension"

"SGIS_filter4_parameters—The Filter4 Parameters Extension"

"SGIS_sharpen_texture—The Sharpen Texture Extension"

"SGIS_texture4D—The 4D Texture Extension"

"SGIS_texture_edge/border_clamp—Texture Clamp Extensions"

"SGIS_texture_filter4—The Texture Filter4 Extensions"

"SGIS_texture_lod—The Texture LOD Extension"

"SGIS_texture_select—The Texture Select Extension"

The following sections describe extensions that are experimental:

"SGIX_clipmap—The Clipmap Extension"

"SGIX_texture_add_env—The Texture Environment Add Extension"

"SGIX_texture_lod_bias—The Texture LOD Bias Extension"

"SGIX_texture_scale_bias—The Texture Scale Bias Extension"

"SGIX_texture_multi_buffer—The Texture Multibuffer Extension"

EXT_texture3D—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 7−1.

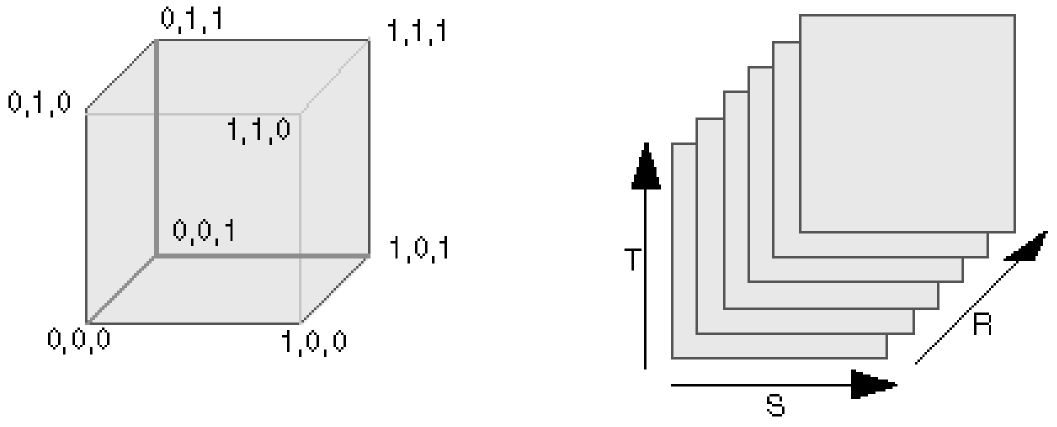


Figure 7−13D Texture

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 plane oriented in 3D space, as shown in Figure 7−2

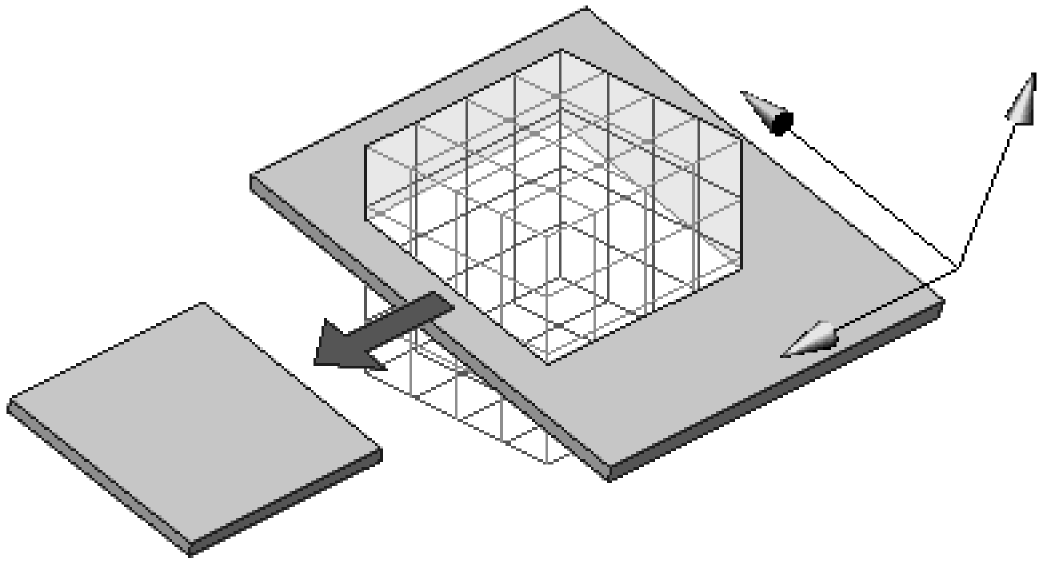


Figure 7−2 Extracting a Planar Texture From a 3D Texture Volume

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()*, which has the following prototype:

```
void glTexImage3DEXT( GLenum target,
                           GLint level,
                           GLenum internalformat,
                           GLsizei width,
                           GLsizei height,
                           GLsizei depth,
                           GLint border,
                           GLenum format,
                           GLenum type,
                           const GLvoid *pixels )
```

The function is defined like *glTexImage2D()* but has a *depth* argument that specifies how many "slices" the texture consists of.

The extension provides the following additional features:

Pixel storage modes. The extension extends the pixel storage modes by adding eight 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_ROWS.

All four modes default to zero.

Texture wrap modes. The functions *glTexParameter*()*, accept the additional token value GL_TEXTURE_WRAP_R_EXT.

GL_TEXTURE_WRAP_R_EXT 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," on page 338 of the *OpenGL Programming Guide*. Mipmapping for 3D textures works the same way: A 3D mipmap is an ordered set of volumes representing the same image; each volume has a resolution lower than the previous one.

The filtering options GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, and GL_LINEAR_MIPMAP_NEAREST, apply to subvolumes instead of subareas. GL_LINEAR_MIPMAP_LINEAR results in two trilinear blends in two different volumes, followed by an LOD blend.

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 "Texture Proxy" on page 330 of the *OpenGL Programming Guide*, *Second Edition.*

You can also call *glGetIntegerv()* with argument GL_MAX_TEXTURE_SIZE_3D_EXT.

Querying. Use the following call to query the 3D texture:

`glGetTexImage(GL_TEXTURE_3D_EXT,` *level*, *format*, *type*, *pixels*`)`

Subvolumes of the 3D texture can be replaced using *glTexSubImage3DEXT()* and *glCopyTexSubImage3DEXT()* (see "Replacing All or Part of a Texture Image," on pages 332 − 335 of the *OpenGL Programming Guide*, *Second Edition*).

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.

Example 7−1 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 windo
w
 * setup, go here.
 */
```

New Functions

glTexImage3DEXT, glTexSubImage3DEXT, glCopyTexImage3DEXT

SGI_texture_color_table—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 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 is 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 with 4 bits per component 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 cannot 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 "SGI_color_table—The Color Table Extension". 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.

Each implementation supports a at least a maximum size of 256 entries. The actual maximum size is implementation−dependent; it is much larger on most Silicon Graphics systems.

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 "Texture Proxy," on page 330 of the *OpenGL Programming Guide*.

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);
}
```

Texture Color Table and Internal Formats

The contents of a texture color table are used to replace a subset of the components of each texel group, based on the base internal format of the table. If the table size is zero, the texture color table is effectively disabled. The texture color table is applied to the texture components Red (Rt), Green (Gt), Blue (Bt), and Alpha(At) texturing components according to the following table:

Table 7−1 Modification of Texture Components

Base Table Internal Format	Rt	Gt	Bt	At
GL_ALPHA	Rt	Gt	Bt	A(At)
GL_LUMINANCE	L(Rt)	L(Gt)	L(Bt)	At
GL_LUMINANCE_ALPHA	L(Rt)	L(Gt)	L(Bt)	A(At)
GL_INTENSITY	I(Rt)	I(Gt)	I(Bt)	I(At)
GL_RGB	R(Rt)	G(Gt)	B(Bt)	At
GL_RGBA	R(Rt)	G(Gt)	B(Bt)	A(At)

Using Texture Color Table On Different Platforms

The texture color table extension is currently implemented on RealityEngine, RealityEngine2, VTX, InfiniteReality, High IMPACT, and Maximum IMPACT systems. For a detailed discussion of machine−dependent issues, see theglColorTableParameterSGI reference page. This section summarizes the most noticeable restrictions.

Texture Color Table on Indigo2 IMPACT Systems

On Indigo2 IMPACT systems, certain combinations of texture internal format and texture color table internal format do not work, as shown in the following table:

Table 7−2 Unsupported Combinations on Indigo2 IMPACT

TCT	Texture
GL_RGB	GL_LUMINANCE or GL_LUMINANCE_ALPHA
GL_RGBA	All formats
GL_INTENSITY	All formats

Texture Color Table on InfiniteReality Systems

InfiniteReality systems reserve an area of 4K 12−bit entries for texture color tables. Applications can use four 1KB tables, two 2KB tables, or one 4KB table. Not all combinations of texture and texture color tables are legal. InfiniteReality systems support the following combinations:

Table 7−3 Supported Combinations on InfiniteReality

TCT size	TCT Format	Texture
>=1024	Any	Any
2048	L, I, LA	L, I, LA
4096	I, L	I, L

SGIS_detail_texture—The Detail Texture Extension

This section discusses the detail texture extension, SGIS_detail_texture, which like the sharpen texture extension (see "SGIS_sharpen_texture—The Sharpen Texture Extension") is useful in situations where you want to maintain good image quality when a texture is 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 "SGIS_sharpen_texture—The Sharpen Texture Extension").

Using the Detail Texture Extension

Because the high−frequency detail in a texture (for example, a road) is often approximately 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 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_RGB
A,
            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 (because 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);
```

   Because 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 7−4 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 7−4 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

Detail Texture Computation

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 7−5, depending on the detail texture mode.

Table 7−5 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 value
DET	detail texture value

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

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

Note that the detail texture level is set on the GL_TEXTURE_2D target, not on GL_DETAIL_TEXTURE_2D_SGIS.

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 7−3shows the default linear interpolation curve and a nonlinear interpolation curve that minimizes aliasing when interpolating between two LODs.

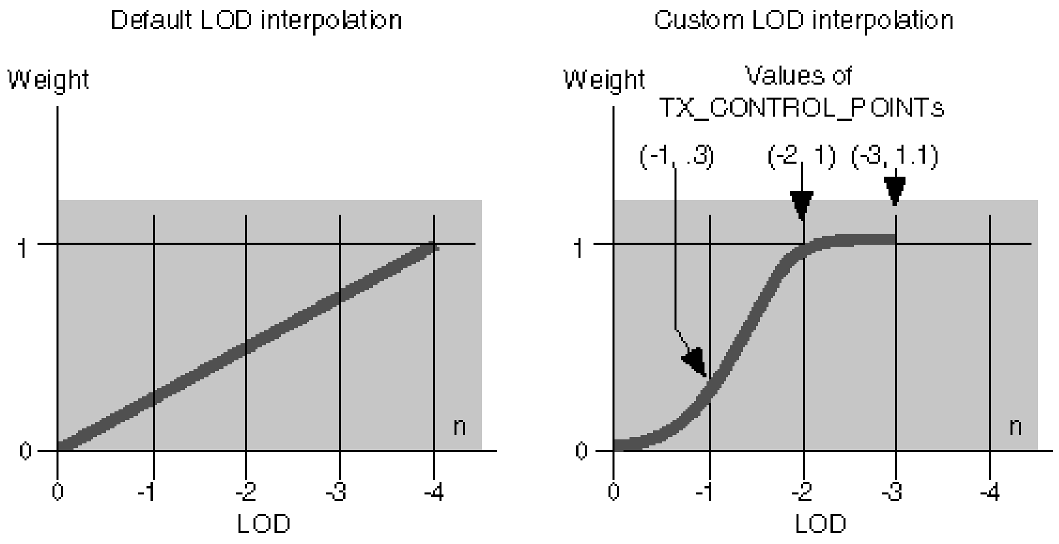


Figure 7−3LOD Interpolation Curves

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. Note that the LOD values are negative.

The following control points can be used to create a nonlinear interpolation function (as shown above in Figure 7−3):

```
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 texture objects, 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. (You cannot bind a detail texture object to GL_TEXTURE_2D.)

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 7−2is a code fragment taken from a simple detail texture example program. The complete example is included in the source tree as *detail.c.* It is also available through the developer toolbox under the same name. For information on toolbox access, see http://www.sgi.com/Technology/toolbox.html.

Example 7−2 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

SGIS_filter4_parameters—The Filter4 Parameters Extension

The filter4 parameters extension, SGIS_filter4_parameters, provides a convenience function that facilitates generation of values needed by the Texture Filter4 extension (see "SGIS_texture_filter4—The Texture Filter4 Extensions").

Note: This extension is part of GLU.

Applications can derive 4 x 4 and 4 x 4 x 4 interpolation coefficients by calculating the cross product of coefficients in 2D or 3D, using the two−pixel−wide span of filter function.

The coefficients are computed in one of two ways:

Using the Mitchell−Netravali scheme. In that case, many of the desired characteristics of other

4x1 interpolation schemes can be accomplished by setting B and C in their piecewise cubic formula. Notably, the blurriness or sharpness of the resulting image can be adjusted with B and C. See Mitchell, Don. and Netravali, Arun, "Reconstruction Filters for Computer Graphics," SIGGRAPH ’88, pp. 221−228.

Using Lagrange interpolation. In that case, four piecewise cubic polynomials (two redundant ones) are used to produce coefficients resulting in images at a high sharpness level. See Dahlquist and Bjorck, "Numerical Methods", Prentice−Hall, 1974, pp 284−285.

To choose one of the two schemas, set the *filtertype* parameter of *gluTexFilterFuncSGI()* to GLU_LAGRANGIAN_SGI or GLU_MITCHELL_NETRAVALI_SGI.

Using the Filter4 Parameters Extension

Applications use the Filter4 Parameter extension in conjunction with the Texture Filter4 extension to generate coefficients that are then used as the *weights* parameter of *glTexFilterFuncSGIS()*.

To generate the coefficients, call *gluTexFilterFuncSGI()* with the following argument values:

target set to GL_TEXTURE_1D or GL_TEXTURE_2D

filterype set to GLU_LAGRANGIAN_SGI or GLU_MITCHELL_NETRAVALI_SGI

params set to the value appropriate for the chosen *filtertype:*

– If *filtertype* is GLU_LAGRANGIAN_SGI, *parms* must be NULL.

– If *filtertype* is GLU_MITCHELL_NETRAVALI_SGI, *parms* may point to a vector of two floats containing B and C control values or *parms* may be NULL in which case both B and C default to 0.5.

n set to a power of two plus one and must be less than or equal to 1025.

weights pointing an array of *n* floating−point values generated by the function. It must point to *n* values of type GL_FLOAT worth of memory.

Note that *gluTexFilterFuncSGI()* and *glTexFilterFuncSGI()* only customize filter4 filtering behavior; texture filter4 functionality needs to be enabled by calling *glTexParameter*()* with *pname* set to TEXTURE_MIN_FILTER or TEXTURE_MAG_FILTER, and *params* set to GL_FILTER4_SGIS. See "Using the Texture Filter4 Extension" for more information.

SGIS_point_line_texgen—The Point or Line Texture Generation Extension

The point or line texgen extension, SGIS_point_line_texgen, adds two texture coordinate generation modes, which both generate a texture coordinate based on the minimum distance from a vertex to a specified line.

The section "Automatic Texture−Coordinate Generation" in Chapter 9, "Texture Mapping" of the *OpenGL Programming Guide, Second Edition*, discusses how applications can use *glTexGen()* to have OpenGL automatically generate texture coordinates.

This extension adds two modes to the existing three. The two new modes are different from the other three. To use them, the application uses one of the newly defined constants for the *pname* parameter and another, matching one for the *param* (or *params*) parameter. For example:

```
glTexGeni(GL_S, GL_EYE_POINT_SGIS, EYE_DISTANCE_TO_POINT_SGIS)
```

Why Use Point or Line Texture Generation

The extension is useful for certain volumetric rendering effects. For example, applications could compute fogging based on distance from an eyepoint.

SGIS_sharpen_texture—The Sharpen Texture Extension

This section discusses the sharpen texture extension, SGIS_sharpen_texture. This extension and the detail texture extension (see "SGIS_detail_texture—The Detail Texture Extension") 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 in the texture image comes from sharp edges, 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 level.

About 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 7−6.

Table 7−6 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 value
LOD1	Texture value 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 is useful to create a nonlinear LOD extrapolation function.

Figure 7−4shows 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.

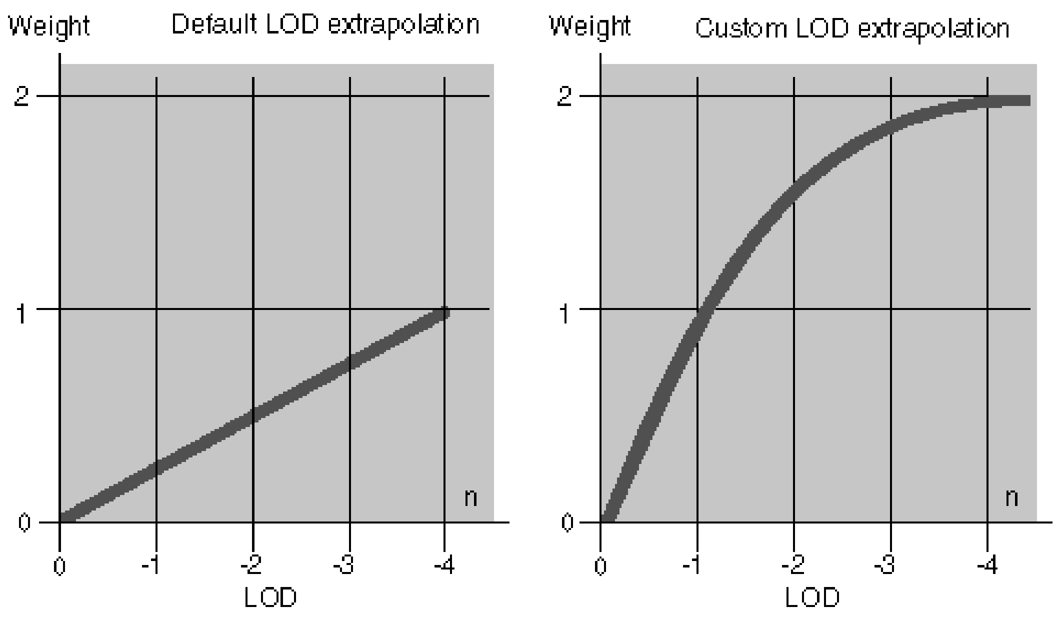


Figure 7−4LOD Extrapolation Curves

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 7−4—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 the kind of function you choose, the function will pass through the control points.

Using Sharpen Texture and Texture Object

If you are using texture objects, 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 7−3illustrates the use of sharpen texture. 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*. It is also available through the developer toolbox under the same name. See http://www.sgi.com/Technology/toolbox.html for information on toolbox access.

Example 7−3 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.

SGIS_texture4D—The 4D Texture Extension

The 4D texture extension, SGIS_texture4D, defines four−dimensional texture mapping. Four−dimensional textures are used primarily as color lookup tables for color conversion.

Note: This extension is currently implemented only on Indigo2 IMPACT and OCTANE systems. Because of that, developers are encouraged to consult information available through the OpenGL home page, most notably the extension specifications.

SGIS_texture_edge/border_clamp—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 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 that want to map a single copy of the texture onto a large surface. Clamping is discussed in detail in the section "Repeating and Clamping Textures" on page 360 of the *OpenGL Programming Guide, Second Edition.*

Why Use the Texture Clamp Extensions?

When a texture coordinate is clamped using the default OpenGL 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 when the texture coordinates are clamped.

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

Both clamping extensions are supported for one−, two−, and three−dimensional textures. Clamping always occurs for texture coordinates less than zero and greater than 1.0.

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

SGIS_texture_filter4—The Texture Filter4 Extensions

The texture filter4 extension, SGIS_texture_filter4, allows applications to filter 1D and 2D textures using an application−defined filter. The filter has to be symmetric and separable and have four samples per dimension. In the most common 2D case, the filter is bicubic. This filtering can yield better−quality images than mipmapping, and is often used in image processing applications.

The *OpenGL Programming Guide, Second Edition*, discusses texture filtering in the section "Filtering" on page 345, as follows: "Texture maps are square or rectangular, but after being mapped to a polygon or surface and transformed into screen coordinates, the individual texels of a texture rarely correspond to individual pixels of the final screen image. Depending on the transformation used and the texture mapping applied, a single pixel on the screen can correspond to anything from a small portion of a texel (magnification) to a large collection of texels (minification)."

Several filters are already part of OpenGL; the extension allows you to define your own custom filter. The custom filter cannot be a mipmapped filter and must be symmetric and separable (in the 2D case).

Using the Texture Filter4 Extension

To use Filter4 filtering, you have to first define the filter function. Filter4 uses an application−defined array of weights (see "Determining the weights Array"). There is an implementation−dependent default set of weights.

Specifying the Filter Function

Applications specify the filter function by calling *glTexFilterFuncSGIS()* (see also the glTexFilterFuncSGIS reference page) with

target set to GL_TEXTURE_1D or GL_TEXTURE_2D

filter set to GL_FILTER4_SGIS

weights pointing to an array of *n* floating−point values. The value*n* must equal 2**m + 1 for some nonnegative integer value of m.

Determining the weights Array

The *weights* array contains samples of the filter function

```
f(x), 0<=x<=2
```

Each element *weights*[i] is the value of

```
f((2*i)/(n−1)), 0<=i<=n−1
```

OpenGL stores and uses the filter function as a set of samples

```
f((2*i)/(Size−1)), 0<=i<=Size−1
```

where *Size* is the implementation−dependent constant GL_TEXTURE_FILTER4_SIZE. If*n* equals *Size*, the array *weights* is stored directly in OpenGL state. Otherwise, an implementation−dependent resampling method is used to compute the stored samples.

Note: "SGIS_filter4_parameters—The Filter4 Parameters Extension" provides interpolation coefficients just as they are required for GL_FILTER4_SGIS filtering.

Size must equal 2**m + 1 for some integer value of *m* greater than or equal to 4. The value *Size* for texture *target* is returned by *params* when *glGetTexParameteriv()* or *glGetTexParameterfv()* is called with *pname* set to TEXTURE_FILTER4_SIZE_SGIS.

Setting Texture Parameters

After the filter function has been defined, call *glTexParameter*()* with

pname set to one of GL_TEXTURE_MIN_FILTER or GL_TEXTURE_MAG_FILTER

param or *params* set to FILTER4_SGIS

the value of *param(s)* set to the function you just defined

Because filter4 filtering is defined only for non−mipmapped textures, there is no difference between its definition for minification and magnification.

New Functions

glTexFilterFuncSGIS, glGetTexFilterFuncSGIS

SGIS_texture_lod—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" on page 338 and "Filtering" on page 344 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 it is used 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 levels after a certain period of time has elapsed. Availability in that case does not depend on what is 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

Note that the number used for the maximum level is absolute, not relative to the base level.

SGIS_texture_select—The Texture Select Extension

The texture select extension, SGIS_texture_select, allows for more efficient use of texture memory by subdividing the internal representation of a texel into one, two, or four smaller texels. The extension may also improve performance of texture loading.

Why Use the Texture Select Extension?

On InfiniteReality graphics systems, the smallest texel supported by the hardware is 16 bits. The extension allows you to pack multiple independent textures together to efficiently fill up space in texture memory (the extension itself refers to each of the independent textures as component groups).

Two eight−bit textures can be packed together. Examples include 8−bit luminance, 8−bit intensity, 8−bit alpha, and 4−bit luminance−alpha.

Four four−bit textures can be packed together. Examples include 4−bit luminance, 4−bit intensity, and 4−bit alpha.

The extension allows developers to work with these components by providing several new texture internal formats. For example, assume that a texture internal format of GL_DUAL_LUMINANCE4_SGIS is specified. Now there are two component groups, where each group has a format of GL_LUMINANCE4. One of the two GL_LUMINANCE groups is always selected. Each component can be selected and interpreted as a GL_LUMINANCE texture.

Note: The point of this extension is to save texture memory. Applications that need only 8−bit or 4−bit texels would otherwise use half or one quarter of texture memory. However, applications that use 16−bit or larger texels (such as RGBA4, LA8) won't benefit from this extension.

Using the Texture Select Extension

To use the texture select extension, first call *glTexImage*D()* to define the texture using one of the new internal formats:

```
glTexImage[n]D[EXT] ( /* Definition */
     internalFormat =
         GL_DUAL_{ ALPHA, LUMINANCE, INTENSITY * }{4, 8, 12, 16 }_SG
IS
         GL_DUAL_LUMINANCE_ALPHA{ 4, 8 } _SGIS
         GL_QUAD_{ ALPHA, LUMINANCE, INTENSITY*}{ 4, 8 }_SGIS
         );
```

The system then assigns parts of the texture data supplied by the application to parts of the 16−bit texel, as illustrated in Table 7−7.

To select one of the component groups for use during rendering, the application then calls *glTexParameter*()* as follows:

```
glTexParameteri ( /* Selection & Usage */
        target = GL_TEXTURE_[n]D[_EXT],
        param = GL_DUAL_TEXTURE_SELECT_SGIS GL_QUAD_TEXTURE_SELECT_S
GIS
        value = { 0, 1 },
                { 0, 1, 2, 3 }
         );
```

There is always a selection defined for both DUAL_TEXTURE_SELECT_SGIS and QUAD_TEXTURE_SELECT_SGIS formats. The selection becomes active when the current texture format becomes one of the DUAL* or QUAD* formats, respectively. If the current texture format is not one of DUAL* or QUAD* formats, this extension has no effect.

Component mapping from the canonical RGBA to the new internal formats is as follows:

Table 7−7 Texture Select Host Format Components Mapping

Format	Grouping
DUAL* formats that are groups of ALPHA, LUMINANCE, and INTENSITY	RED component goes to the first group ALPHA component goes to the second group
DUAL* formats that are groups of LUMINANCE_ALPHA	RED and GREEN components go to the first group BLUE and ALPHA go to the second group
QUAD* formats	RED component goes to the first group GREEN component to the second group BLUE component to the third group ALPHA component to the fourth group

The interpretation of the bit resolutions of the new internal formats is implementation dependent. To query the actual resolution that is granted, call *glGetTexLevelParameter()* with *pname* set appropriately, for example GL_TEXTURE_LUMINANCE_SIZE. The bit resolution of similar type components in a group, such as multiple LUMINANCE components, is always the same.

SGIX_clipmap—The Clipmap Extension

The clipmap extension, SGIX_clipmap, allows applications to use dynamic texture representations that efficiently cache textures of arbitrarily large size in a finite amount of physical texture memory. Only those parts of the mipmapped texture that are visible from a given application−specified location are stored in system and texture memory. As a result, applications can display textures too large to fit in texture memory by loading parts on the texture into texture memory only when they are required.

Full clipmap support is implemented in IRIS Performer 2.2 (or later). Applications can also use this extension on the appropriate hardware (currently InfiniteReality only) for the same results. In that case, the application has to perform memory management and texture loading explicitly.

This section explains how clipmaps work and how to use them in the following sections:

"Clipmap Overview" explains the basic assumptions behind clipmaps.

"Using Clipmaps From OpenGL" provides step by step instructions for setting up a clipmap stack and for using clipmaps. Emphasis is on the steps, with references to the background information as needed.

"Clipmap Background Information" explains some of the concepts behind the steps in clipmap creation in more detail.

"Virtual Clipmaps" discusses how to work with a virtualized clipmap, which is the appropriate solution if not all levels of the clipmap fit.

Note: For additional conceptual information, see the specification for the clipmap extension, which is available through the developer's toolbox.

Clipmap Overview

Clipmaps avoid the size limitations of normal mipmaps by clipping the size of each level of a mipmap texture to a fixed area, called the clip region (see Figure 7−5). A mipmap contains a range of levels, each four times the size of the previous one. Each level (size) determines whether clipping occurs:

For levels smaller than the clip region—that is, for low−resolution levels that have relatively few texels—the entire level is kept in texture memory.

Levels larger than the clip region are clipped to the clip region's size. The clip region is set by the application, trading off texture memory consumption against image quality. (The image may become blurry because texture accesses outside the clip region are forced to use a coarse LOD.)

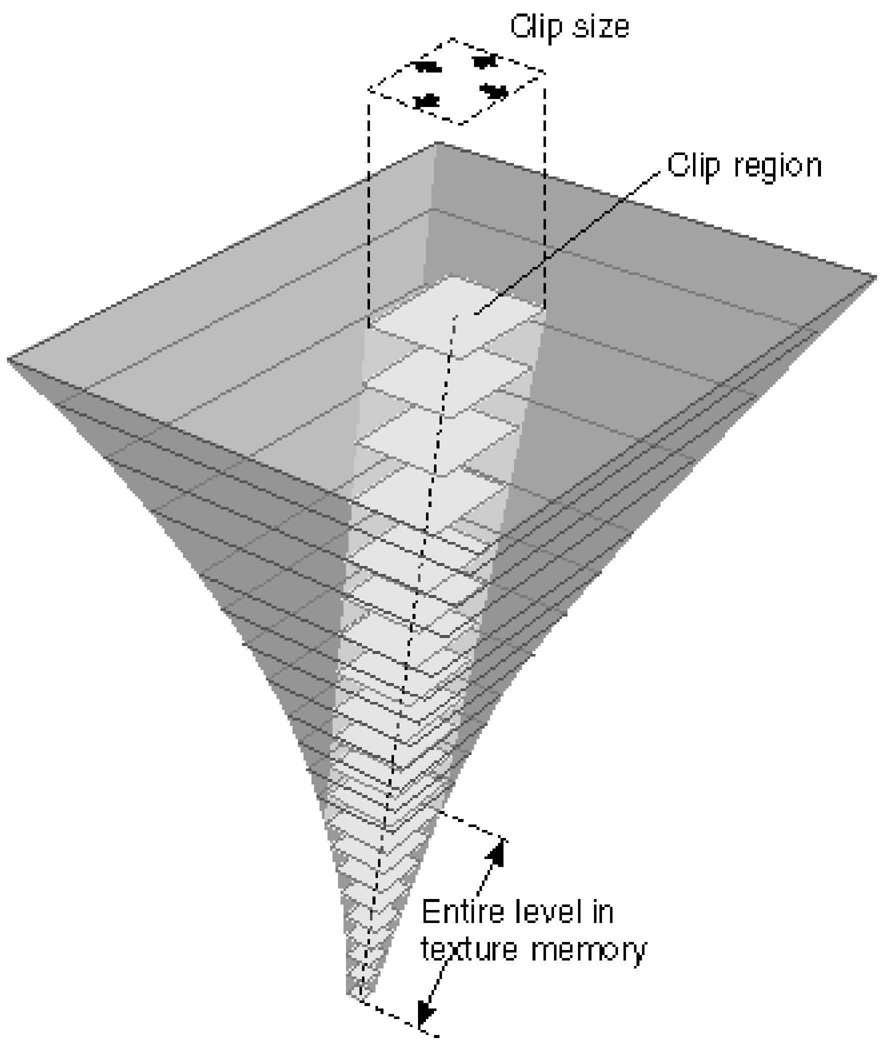


Figure 7−5 Clipmap Component Diagram

Clipmap Constraints

The clipmap algorithm is based on the following constraints:

The viewer can see only a small part of a large texture from any given viewpoint.

The viewer looks at a texture from only one location.

The viewer moves smoothly relative to the clipmap geometry (no teleporting).

The textured geometry must have a reasonable, relatively flat topology.

Given these constraints, applications can maintain a high−resolution texture by keeping only those parts of the texture closest to the viewer in texture memory. The remainder of the texture is on disk and cached in system memory.

Why Do the Clipmap Constraints Work?

The clipmap constraints work because only the textured geometry closest to the viewer needs a high−resolution texture. Distant objects are smaller on the screen, so the texels used on that object also appear smaller (cover a small screen area). In normal mipmapping, coarser mipmap levels are chosen as the texel size gets smaller relative to the pixel size. These coarser levels contain fewer texels because each texel covers a larger area on the textured geometry.

Clipmaps store only part of each large (high−resolution) mipmap level in texture memory. When the user looks over the geometry, the mipmap algorithm starts choosing texels from a lower level before running out of texels on the clipped level. Because coarser levels have texels that cover a larger area, at a great enough distance, texels from the unclipped, smaller levels are chosen as appropriate.

When a clip size is chosen, the mipmap levels are separated into two categories:

Clipped levels, which are texture levels that are larger than the clip size.

Nonclipped levels, which are small enough to fit entirely within the clip region.

The nonclipped levels are viewpoint independent; each nonclipped texture level is complete. Clipped levels, however, must be updated as the viewer moves relative to the textured geometry.

Clipmap Textures and Plain Textures

Clipmaps are not completely interchangeable with regular OpenGL textures. Here are some differences:

Centering. In a regular texture, every level is complete in a regular texture. Clipmaps have clipped levels, where only the portion of the level near the clipmap center is complete. In order to look correct, a clipmap center must be updated as the viewport of the textured geometry moves relative to the clipmap geometry.

As a result, clipmaps require functionality that recalculates the center position whenever the viewer moves (essentially each frame). This means that the application has to update the location of the clip center as necessary.

Texel Data. A regular texture is usually only loaded once, when the texture is created. The texel data of a clipmap must be updated by the application each time the clipmap center is moved. This is usually done by calling *glTexSubImage2D()* and using the toroidal loading technique (see "Toroidal Loading").

Using Clipmaps From OpenGL

To use clipmaps, an application has to take care of two distinct tasks, discussed in this section:

"Setting Up the Clipmap Stack"

"Updating the Clipmap Stack"

Setting Up the Clipmap Stack

To set up the clipmap stack, an application has to follow these steps:

1. Call *glTexParameter*()* with the GL_TEXTURE_MIN_FILTER_SGIX parameter set to GL_LINEAR_CLIPMAP_LINEAR_SGIX to let OpenGL know that clipmaps, not mipmaps will be used.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR_CLIPMAP_LINEAR);
```

GL_TEXTURE_MAG_FILTER can be anything but GL_FILTER4_SGIS

2. Set the GL_TEXTURE_CLIPMAP_FRAME_SGIX parameter to set an invalid border region of at least eight pixels.

   The frame is the part of the clip that the hardware should ignore. Using the frame avoids certain sampling problems; in addition, the application can load into the Frame region while updating the texture. See "Invalid Borders" for more information.

   In the following code fragment, *size* is the fraction of the clip size that should be part of the border; that is, .2 would mean 20 percent of the entire clip size area would be dedicated to the invalid border, along the edge of the square clip size region.

```
GLfloat size = .2f;      /* 20% */
/* can range from 0 (no border) to 1 (all border) */
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_CLIPMAP_FRAME_SGIX,size)
;
```

3. Set GL_TEXTURE_CLIPMAP_CENTER_SGIX to set the center texel of the highest−resolution texture, specified as an integer. The clip center is specified in terms of the top (highest−resolution) level of the clipmap, level 0. OpenGL automatically adjusts and applies the parameters to all of the other levels.

   The position of the center is specified in texel coordinates. Texel coordinate are calculated by taking the texture coordinates (which range from 0 to 1 over the texture) and multiplying them by the size of the clipmap’s top level. See "Moving the Clip Center" for more information.

   The following code fragment specifies the location of the region of interest on every clipped level of clipmap. The location is specified in texel coordinates, so texture coordinates must be multiplied by the size of the top level in each dimension. In this example, *center* is at the center of texture (.5, .5). Assume this clipmap is 4096 (s direction) by 8192 (t direction) at level 0.

```
  int center[3];
  center[0] = .5 * 4096;
  center[1] = .5 * 8192;
  center[2] = 0; /* always zero until 3d clipmaps supported */

  glTexParameteriv(GL_TEXTURE_2D, GL_TEXTURE_CLIPMAP_CENTER_SGIX,
center);
```

4. Set GL_TEXTURE_CLIPMAP_OFFSET_SGIX to specify the offset. The *offset* parameter allows applications to offset the origin of the texture coordinates so that the incrementally updated texture appears whole and contiguous.

   Like the center, the offset is supplied in texel coordinates. In the code fragment below, clip size is the size of the region of interest.

```
  int offset[2];

  offset[0] = (center[0] + clipsize/2) % clipsize;
  offset[1] = (center[1] + clipsize/2) % clipsize;
```

```
glTexParameteriv(GL_TEXTURE_2D,
   GL_TEXTURE_CLIPMAP_OFFSET_SGIX,
   offset);
```

5. Call *glTexImage2D()* to define the highest−resolution level that contains the entire map. This indirectly tells OpenGL what the clip size is and which level of the clipmap contains the largest clipped level. OpenGL indirectly calculates the clip size of a clipmap by the size of the texture levels. Although the clipmap levels can be loaded in any order, it is most efficient for the current clipmap system if the top of the pyramid is loaded first. Note that a clipmap's clip size level is at some level other than zero (otherwise there would be no levels larger than the clip size; that is, no clipped levels.)

   In the following code fragment, the clipmap is RGB, with a top level of dimensions 8192 by 8192, and a clip size of 512 by 512. There will be 12 levels total, and the last level at which the whole mipmap is in memory (512 level) is level 4.

```
GLint pyramid_level, border = 0;
GLsizei clipsize_wid, clipsize_ht;
 clipsize_wid = clipsize_ht = 512;
pyramid_level = 4; /* 8192 = 0, 4096 = 1, 2048 = 2, 1024 = 3, ...
 */

  glTexImage2D(GL_TEXTURE_2D,
        pyramid_level,
        GL_RGB, /* internal format */
        clipsize_wid,
        clipsize_ht,
        border, /* not invalid border! */,
        GL_RGB, /* format of data being loaded */
        GL_BYTE, /* type of data being loaded */
        data); /* data can be null and subloaded later if desired
 */
```

6. Create the clipmap stack by calling *glTexImage2D()* repeatedly for each level.

   If you want to use a virtual clipmap, you can use the texture_LOD extension (see "SGIS_texture_lod—The Texture LOD Extension") to specify the minimum and maximum LOD. See "Virtual Clipmaps".

7. After the application has precomputed all mipmaps, it stores them on disk for easy access. Note that it is not usually possible to create the stack in real time.

Updating the Clipmap Stack

As the user moves through the "world," the center of the clipmap usually changes with each frame. Applications therefore have to update the clipmap stack with each frame, following these steps:

1. Compute the difference between the old and new center.

   See "Moving the Clip Center" for background information.

2. Determine the incremental texture load operations needed for each level.

3. Perform toroidal loads by calling *glTexSubImage2D()* to load the appropriate texel regions.

   "Toroidal Loading" discusses this in more detail.
4. Set the parameters for center and offset for the next move.

Clipmap Background Information

The following sections provide background information for the steps in "Using Clipmaps From OpenGL".

Moving the Clip Center

Only a small part of each clipped level of a clipmap actually resides in texture memory. As a result, moving the clip center requires updating the contents of texture memory so it contains the pixel data corresponding to the new location of the region of interest.

Updates must usually happen every frame, as shown in Figure 7−6 Applications can update the clipmaps to the new center using toroidal loading (see "Toroidal Loading").

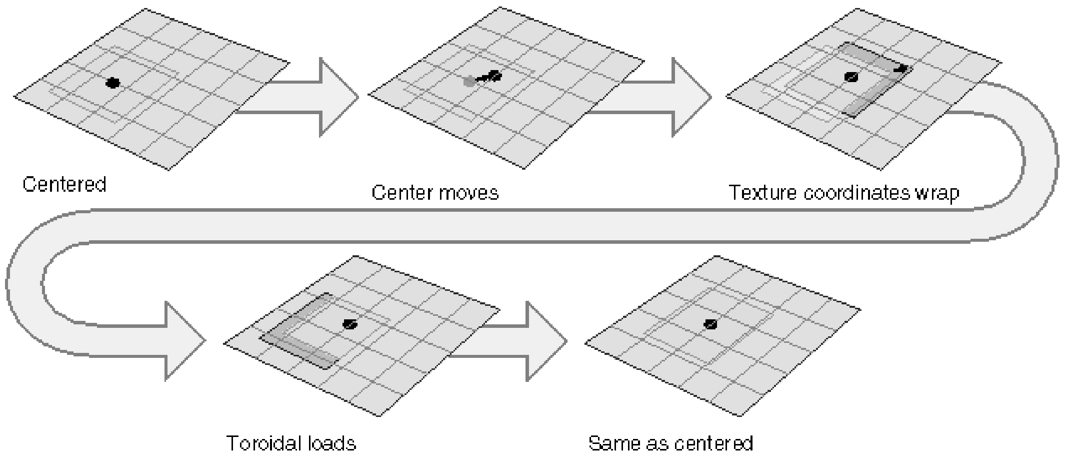


Figure 7−6 Moving the Clip Center

The clip center is set by the application for level 0, the level with the highest resolution. The clipmap code has to derive the clip center location on all levels. As the viewer roams over a clipmap, the centers of each mipmap level move at a different rate. For example, moving the clip center one unit corresponds to the center moving one half that distance in each dimension in the next−coarser mipmap level.

When applications use clipmaps, most of the work consists of updating the center properly and updating the texture data in the clipped levels reliably and efficiently for each frame.To facilitate loading only portions of the texture at a time, the texture data should first be subdivided into a contiguous set of rectangular areas, called *tiles*. These tiles can then be loaded individually from disk into texture memory.

Invalid Borders

Applications can improve performance by imposing alignment requirements to the regions being downloaded to texture memory. Clipmaps support the concept of an *invalid border* to provide this feature. The border is an area around the perimeter of a clip region that is guaranteed not to be displayed. The invalid border shrinks the usable area of the clip region, and can be used to dynamically change the effective size of the clip region.

When texturing requires texels from a portion of an invalid border at a given mipmap level, the texturing system moves down a level, and tries again. It keeps going down to coarser levels until it finds texels at the proper coordinates that are not in the invalid region. This is always guaranteed to happen, because each level covers the same area with fewer texels. Even if the required texel is clipped out of every clipped level, the unclipped pyramid levels will contain it.

The invalid border forces the use of lower levels of the mipmap. As a result, it

Reduces the abrupt discontinuity between mipmap levels if the clip region is small.

Using coarser LODs blends mipmap levels over a larger textured region.

Improves performance when a texture must be roamed very quickly.

Because the invalid border can be adjusted dynamically, it can reduce the texture and system memory loading requirements at the expense of a blurrier textured image.

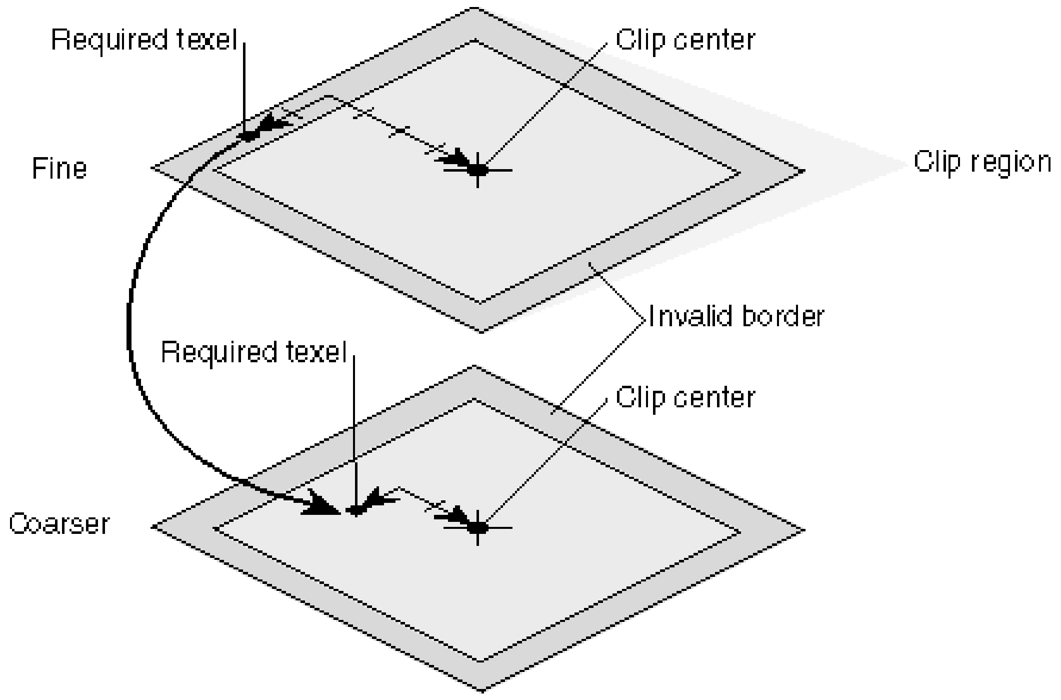


Figure 7−7 Invalid Border

Toroidal Loading

To minimize the bandwidth required to download texels from system to texture memory, the image cache’s texture memory should be updated using *toroidal loading*, which means the texture wraps upon itself. (see Figure 7−6).

A toroidal load assumes that changes in the contents of the clip region are incremental, such that the update consists of

new texels that need to be loaded

texels that are no longer valid

texels that are still in the clip region, but have shifted position

Toroidal loading minimizes texture downloading by updating only the part of the texture region that needs new texels. Shifting texels that remain visible is not necessary, because the coordinates of the clip region wrap around to the opposite side.

As the center moves, only texels along the edges of the clipmap levels change. To allow for incremental loading only of these texels via *glTexSubImage2D()*, toroidal offset values have to be added to the texture addresses of each level. The offset is specified by the application (see "Setting Up the Clipmap Stack"). The offsets for the top level define the offsets for subsequent levels by a simple shift, just as with the center.

Virtual Clipmaps

You can use the texture LOD extension in conjunction with mipmapping to change the base level from zero to something else. Using different base levels results in clipmaps with more levels than the hardware can store at once when texturing.

These larger mipmapped textures can be used by only accessing a subset of all available mipmap levels in texture memory at any one time. A virtual offset is used to set a virtual "level 0" in the mipmap, while the number of effective levels indicates how many levels starting from the new level 0 can be accessed. The minLOD and maxLOD are also used to ensure that only valid levels are accessed. The application typically divides the clipmapped terrain into pieces, and sets the values as each piece is traversed, using the relative position of the viewer and the terrain to calculate the values.

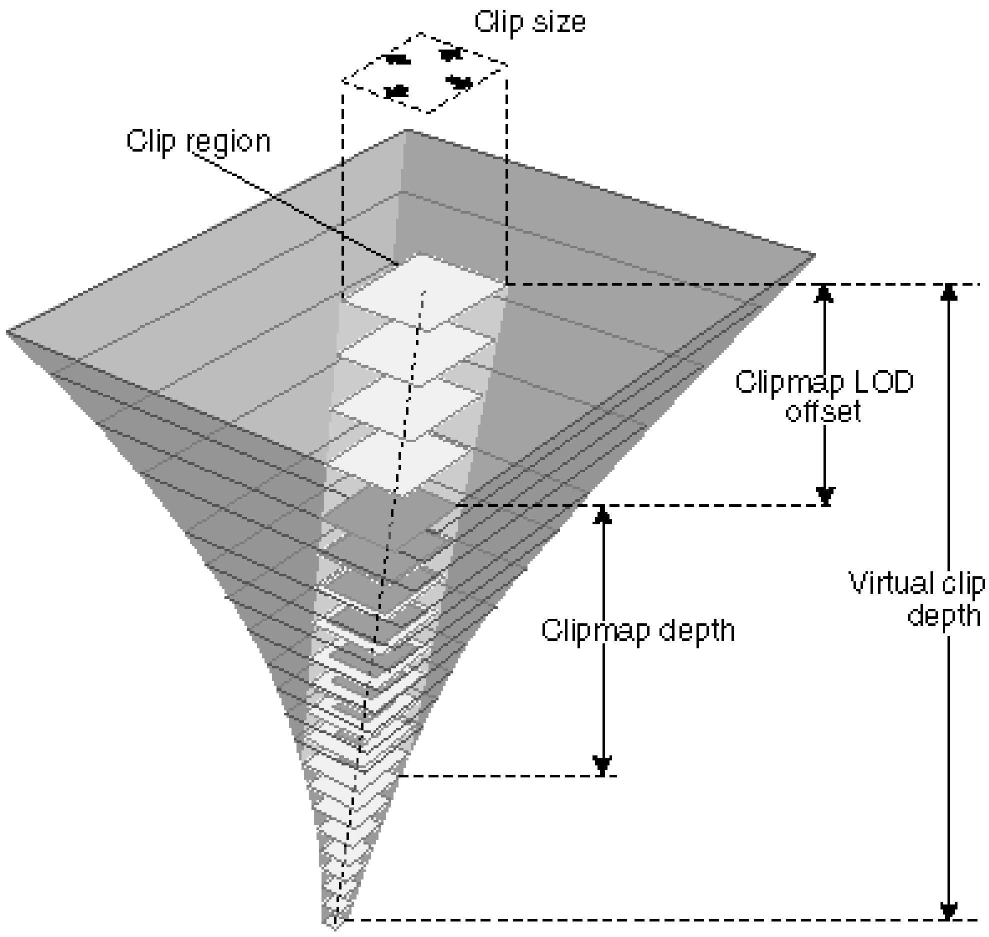


Figure 7−8 Virtual Clipmap

To index into a clipmap of greater than GL_MAX_CLIPMAP_DEPTH_SGIX levels of detail, additional parameters are provided to restrictively index a smaller clipmap of (N+1) levels located wholly within a complete, larger clipmap. Figure 7−8illustrates how a virtual clipmap fits into a larger clipmap stack. The clipmap extension specification explains the requirements for the larger and smaller clipmap in more detail.

When creating a virtual clipmap, an application calls *glTexParameteriv()*, or *glTexParameterfv()* with

target set to GL_TEXTURE_2D

pname set to GL_TEXTURE_CLIPMAP_VIRTUAL_DEPTH_SGIX

params set to (D,N+1,V+1)

where D is the finest level of the clipmap, N+1 is the depth of the clipmap, and V+1 is the depth of the virtual clipmap.

If the depth of the virtual clipmap is zero, clipmap virtualization is ignored, and texturing proceeds as with a non−virtual clipmap.

If you have virtualized the clipmap, you will be adjusting the LOD offset and possibly the number of displayable levels as you render each chunk of polygons that need a different set of clipmap levels to be rendered properly. The application has to compute the levels needed.

SGIX_texture_add_env—The Texture Environment Add Extension

The texture environment add extension, SGIX_texture_add_env, defines a new texture environment function, which scales the texture values by the constant texture environment color, adds a constant environment bias color, and finally adds the resulting texture value on the in−coming fragment color. The extension can be used to simulate highlights on textures (although that functionality is usually achieved with multi−pass rendering) and for situations in which it has to be possible to make the existing color darker or lighter, for example, for simulating an infrared display in a flight simulator.

OpenGL 1.1 supports four texture environment functions: GL_DECAL, GL_REPLACE, GL_MODULATE, and GL_BLEND.

The extension provides an additional environment, GL_ADD, which is supported with the following equation:

```
Cv = Cf + CcCt + Cb
```

where

Cr	Fragment color
Cc	Constant color (set by calling *glTexEnv()*) with pname set to GL_TEXTURE_ENV_COLOR)
Ct	Texture color
Cb	Bias color (set by calling *glTexEnv()* with *pname* set to GL_TEXTURE_ENV_BIAS_SGIX.) and *param* set to a value greater than −1 and less than 1.

The new function works just like the other functions discussed in the section "Texture Functions" on page 354 of the *OpenGL Programming Guide, Second Edition*.

SGIX_texture_lod_bias—The Texture LOD Bias Extension

The texture LOD bias extension, SGIX_texture_lod_bias, allows applications to bias the default LOD to make the resulting image sharper or more blurry. This can improve image quality if the default LOD is not appropriate for the situation in question.

Background: Texture Maps and LODs

If an application uses an image as a texture map, the image may have to be scaled down to a smaller size on the screen. During this process the image must be filtered to produce a high−quality result. Nearest−neighbor or linear filtering do not work well when an image is scaled down; for better results, an OpenGL program can use mipmapping. A mipmap is a series of prefiltered texture maps of decreasing resolution. Each texture map is referred to as one level of detail or LOD. Applications create a mipmap using the routines *gluBuild1DMipmaps()* or *gluBuild2DMipmaps()*. Mipmaps are discussed starting on page 338 of the *OpenGL Programming Guide, Second Edition*.

Graphics systems from Silicon Graphics automatically select an LOD for each textured pixel on the screen. However, in some situations the selected LOD results in an image that is too crisp or too blurry for the needs of the application. For example, 2D mipmapping works best when the shape of the texture on the screen is a square. If that is not the case, then one dimension of the texture must be scaled down more than the other to fit on the screen. By default the LOD corresponding to the larger scale factor is used, so the dimension with the smaller scale factor will appear too blurry.

Figure 7−9shows an image that is too blurry with the default LOD bias. You can see that the marker in the middle of the road is blurred out. In Figure 7−10, this effect is exaggerated by a positive LOD bias. Figure 7−11shows how the markers become visible with a negative LOD bias.

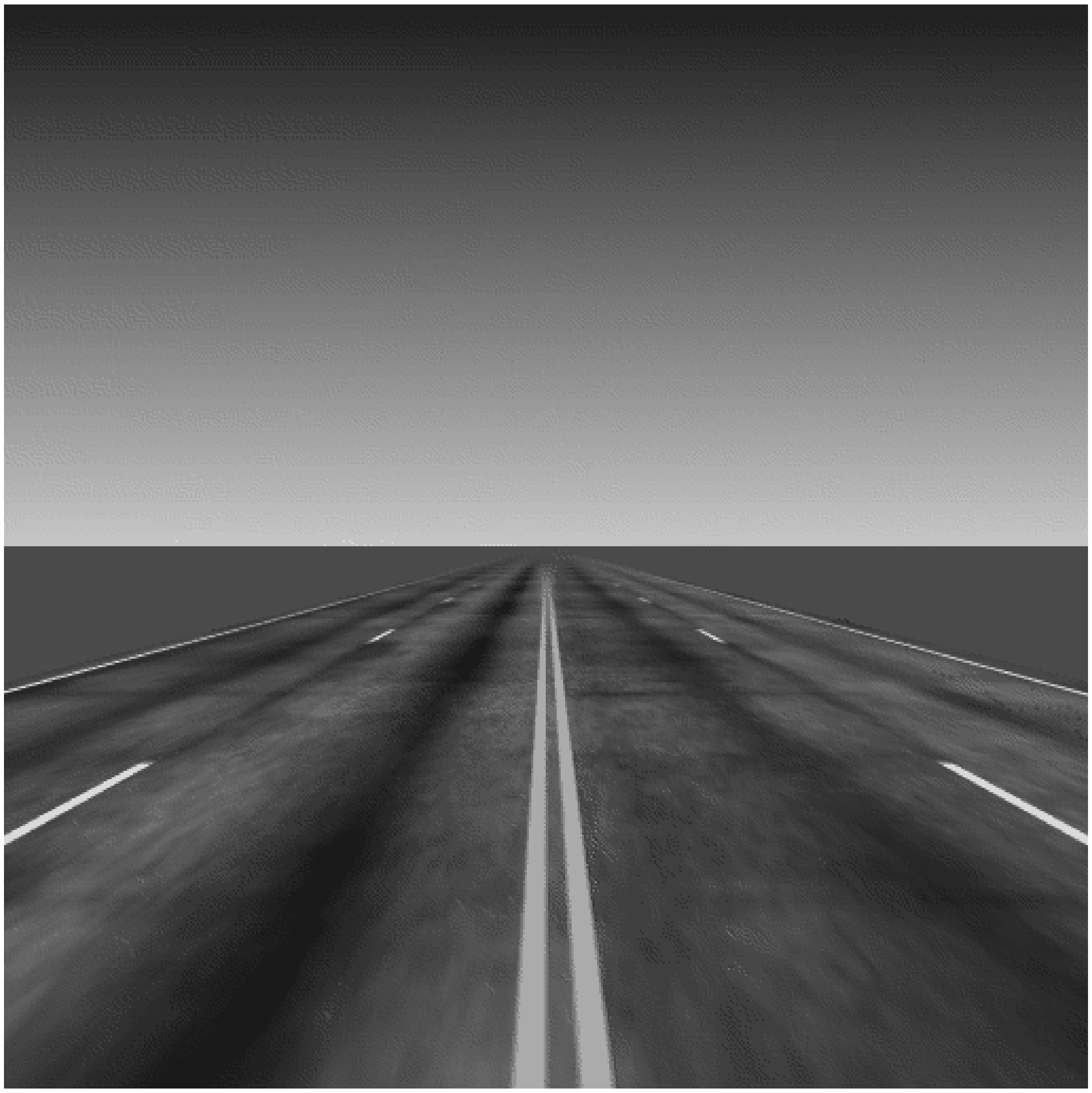


Figure 7−9 Original Image

Figure 7−10 Image With Positive LOD Bias

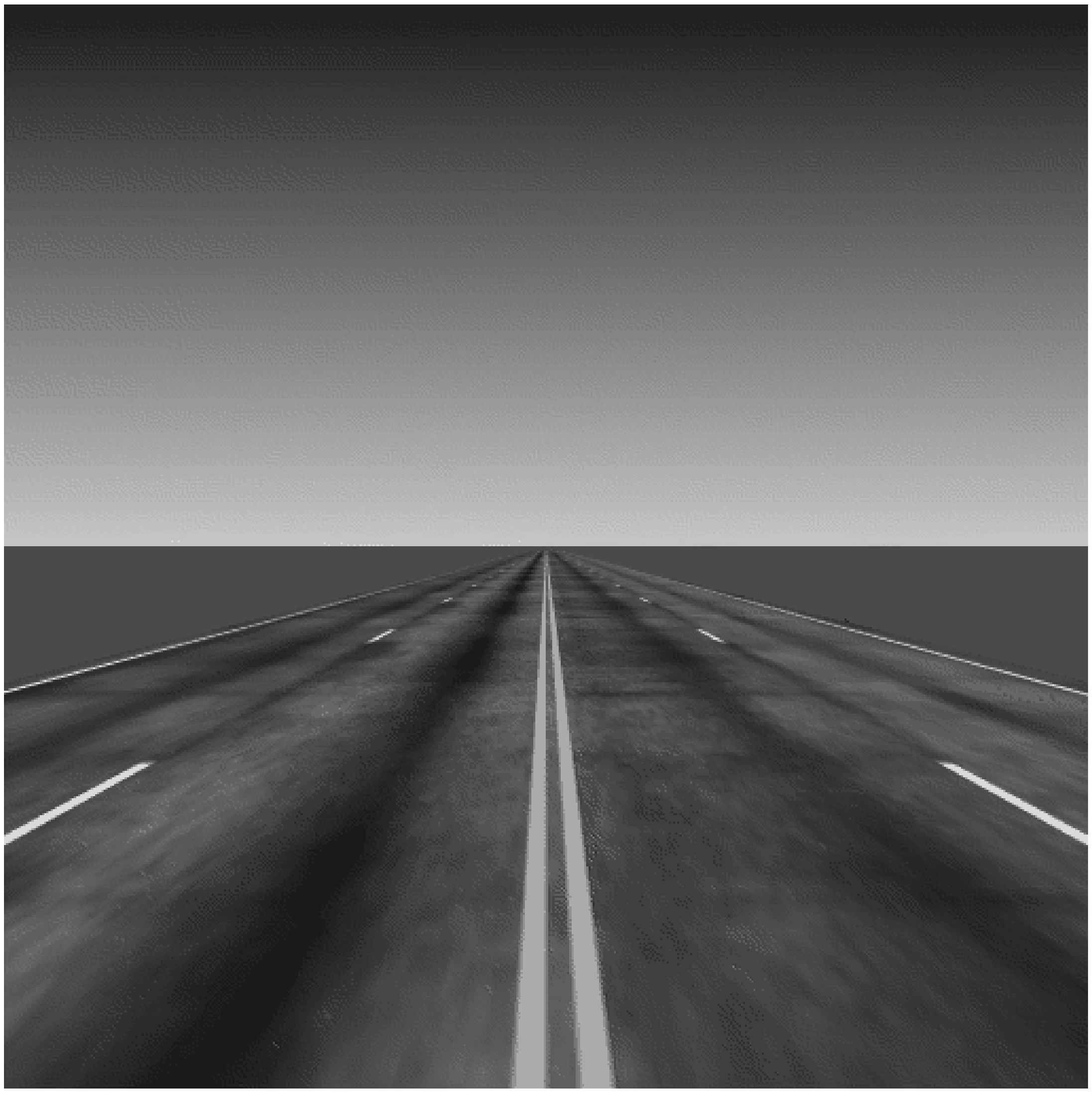


Figure 7−11 Image with Negative LOD Bias

As another example, the texture data supplied by the application may be slightly oversampled or undersampled, so the textured pixels drawn on the screen may be correspondingly blurry or crisp.

Why Use the LOD Bias Extension?

The texture LOD bias extension allows applications to bias the default LOD to make the resulting image sharper or more blurry. An LOD of 0 corresponds to the most−detailed texture map, an LOD of 1 corresponds to the next smaller texture map, and so on. The default bias is zero, but if the application specifies a new bias, that bias will be added to the selected LOD. A positive bias produces a blurrier image, and a negative bias produces a crisper image. A different bias can be used for each dimension of the texture to compensate for unequal sampling rates.

Examples of textures that can benefit from this LOD control include:

Images captured from a video source. Because video systems use non−square pixels, the horizontal and vertical dimensions may require different filtering.

A texture that appears blurry because it is mapped with a nonuniform scale, such as a texture for a road or runway disappearing toward the horizon (the vertical dimension must be scaled down a lot near the horizon, the horizontal dimension is not scaled down much).

Textures that don’t have power of two dimensions and therefore had to be magnified before mipmapping (the magnification may have resulted in a nonuniform scale).

Using the Texture LOD Bias Extension

To make a mipmapped texture sharper or blurrier, applications can supply a negative or positive bias by calling *glTexParameter*()* with

target set to TEXTURE_1D, TEXTURE_2D, or TEXTURE_3D_EXT.

pname set to GL_TEXTURE_LOD_BIAS_S_SGIX, GL_TEXTURE_LOD_BIAS_T_SGIX, or GL_TEXTURE_LOD_BIAS_R_SGIX.

param set to (or *params* pointing to) the desired bias value, which may be any integer or floating−point number. The default value is 0.

You can specify a bias independently for one or more texture dimensions. The final LOD is at least as large as the maximum LOD for any dimension; that is, the texture is scaled down by the largest scale factor, even though the best scale factors for each dimension may not be equal.

Applications can also call *glGetTexParameter*()* to check whether one of these values has been set.

SGIX_texture_scale_bias—The Texture Scale Bias Extension

The texture_scale_bias extension, SGIX_texture_scale_bias, allows applications to perform scale, bias, and clamp operations as part of the texture pipeline. By allowing scale or bias operations on texels, applications can make better utilization of the color resolution of a particular texture internal format, by, for example, performing histogram normalization, or gamut expansion. In addition some color remapping may be performed with this extension if a texture color lookup table is not available or too expensive.

The scale, bias, and clamp operations are applied, in that order, directly before the texture environment equations, or, if the SGI_texture_color_table extension exists, directly before the texture color lookup table. The four values for scale (or bias) correspond to the R, G, B, and A scale (or bias) factors. These values are applied to the corresponding texture components, Rt, Gt, Bt, and At. Following the scale and bias is a clamp to the range [0, 1].

To use the extension, an application calls *glTexParameter*()* with a *pname* parameter GL_POST_TEXTURE_FILTER_BIAS_SGIX or GL_POST_TEXTURE_FILTER_SCALE_SGIX and with *params* set to an array of four values.The scale or bias values can be queried using *glGetTexParameterfv()* or *glGetTexParameteriv()*. The scale, bias, and clamp operations are effectively disabled by setting the four scale values to 1 and the four bias values to 0. There is no specific enable or disable token for this extension.

Because an implementation may have a limited range for the values of scale and bias (for example, due to hardware constraints), this range can be queried. To obtain the scale or bias range, call *glGet*()* with GL_POST_TEXTURE_FILTER_SCALE_RANGE_SGIX or GL_POST_TEXTURE_FILTER_BIAS_RANGE_SGIX, respectively as the *value* parameter. An array of two values is returned: the first is the minimum value and the second is the maximum value.

SGIX_texture_multi_buffer—The Texture Multibuffer Extension

The texture multibuffer extension, SGIX_texture_multi_buffer, allows applications to change the way OpenGL handles multiple textures.

Texture objects, which were introduced in OpenGL 1.1, allow the simultaneous definition of multiple textures. As a result, you can in principle render one texture and at the same time load another texture into hardware or perform other actions on its definition. This is true as long as all redefinitions strictly follow any use of the previous definition.

Conceptually using textures in this fashion is similar to frame buffer double−buffering, except that the intent here is to provide a hint to OpenGL to promote such double−buffering if and wherever possible. The effect of such a hint is to speed up operations without affecting the result. Developers on any particular system must be knowledgable and prepared to accept any trade−offs that may result from such a hint.

The extension is currently used for video texture−mapping; that is, instead of mapping a static image onto an object in a 3D view, live video is mapped. So there is a variety of special effects that can be done. On Indigo2 IMPACT and OCTANE, the method is to use a GLX extension to set the "readsource" to be "video" and then call *glCopyTexImage2D()* to get the latest video image into texture memory. Using the multibuffer extension, it is possible to be drawing with the previous video frame (the front buffer) while the new frame is being loaded in (the back buffer). This really speeds things up.

How to use the Texture Multibuffer Extension

To use the extension, call *glHint()* with the *target* parameter set to GL_TEXTURE_MULTI_BUFFER_HINT_SGIX.

If you specify a hint of GL_FASTEST, texture multi−buffering is used whenever possible to improve performance. Generally, textures that are adjacent in a sequence of multiple texture definitions have the greatest chance of being in different buffers. The number of buffers available at any time depends on various factors, such as the machine being used and the textures' internal formats.

Chapter 8
Rendering Extensions

This chapter explains how to use the different OpenGL rendering extensions. Rendering refers to several parts of the OpenGL pipeline: the evaluator stage, rasterization, and per−fragment operations. You learn about

"Blending Extensions"

"SGIS_fog_function—The Fog Function Extension"

"SGIS_fog_offset—The Fog Offset Extension"

"SGIS_multisample—The Multisample Extension"

"SGIS_point_parameters—The Point Parameters Extension"

"SGIX_reference_plane—The Reference Plane Extension"

"SGIX_shadow, SGIX_depth_texture, and SGIX_shadow_ambient—The Shadow Extensions"

"SGIX_sprite—The Sprite Extension"

Blending Extensions

Blending refers to the process of combining color values from an incoming pixel fragment (a source) with current values of the stored pixel in the framebuffer (the destination). The final effect is that parts of a scene appear translucent. 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, Fog, and Polygon Offset" of the *OpenGL Programming Guide*. The section, which starts on page 214, 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:

"Constant Color Blending Extension"

"Minmax Blending Extension"

"Blend Subtract Extension"

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 were 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 8−1 as source or destination factor, or both.

Table 8−1 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_EX
T);
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

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 intensity projection (MIP) 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 do *not* use source or destination factors, that is, the values specified with *glBlendFunc()* do not apply.

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 "Blend Subtract Extension". 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

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.

SGIS_fog_function—The Fog Function Extension

Standard OpenGL defines three fog modes; GL_LINEAR, GL_EXP (exponential), and GL_EXP2 (exponential squared). Visual simulation systems can benefit from more sophisticated atmospheric effects, such as those provided by the fog function extension.

Note: The fog function extension is supported only on InfiniteReality systems.

The fog function extension, SGIS_fog_function, allows you to define an application−specific fog blend factor function. The function is defined by a set of *control points* and should be monotonic. Each control point is represented as a pair of the eye−space distance value and the corresponding value of the fog blending factor. The minimum number of control points is 1. The maximum number is implementation dependent.

To specify the function for computing the blending factor, call *glFogFuncSGIS()* with *points* pointing at an array of pairs of floating point values, and *n* set to the number of value pairs in *points.* The first value of each value pair in *points* specifies a value of eye−space distance (should be nonnegative), and the second value of each value pair specifies the corresponding value of the fog blend factor (should be in the [0.0, 1.0] range). If there is more than one point, the order in which the points are specified is based on the following requirements:

> The distance value of each point is not smaller than the distance value of its predecessor.
>
> The fog factor value of each point is not bigger than the fog factor value of its predecessor.

The *n* value pairs in *points* completely specify the function, replacing any previous specification that may have existed. At least one control point should be specified. The maximum number of control points is implementation dependent and may be retrieved by *glGet*()* commands.

Initially the fog function is defined by a single point (0.0, 1.0). The fog factor function is evaluated by fitting a curve through the points specified by *glFogFuncSGIS()*. This curve may be linear between adjacent points, or it may be smoothed, but it will pass exactly through the points, limited only by the resolution of the implementation. The value pair with the lowest distance value specifies the fog function value for all values of distance less than or equal to that pair's distance. Likewise, the value pair with the greatest distance value specifies the function value for all values of distance greater than or equal to that pair's distance.

If *pname* is GL_FOG_MODE and *param* is, or *params* points to an integer GL_FOG_FUNC_SGIS, then the application−specified fog factor function is selected for the fog calculation.

FogFunc Example Program

The following simple example program for fog−function extension can be executed well only on those platforms where the extension is supported (currently InfiniteReality only).

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

/* Simple demo program for fog−function. Will work only on machines
 * where SGIS_fog_func is supported (InfiniteReality).
 *
 * Press ‘f’ key to toggle between fog and no fog
 * Pres ESC to quit
 *
 * cc fogfunc.c −o fogfunc −lglut −lGLU −lGL −lXmu −lX11
 */

#define ESC 27

GLint width = 512, height = 512;
GLint dofog = 1;                            /* fog enabled by defaul
t */
```

```
GLfloat fogfunc[] = {                       /* fog−function profile
*/
  6.0, 1.0,   /* (distance, blend−factor) pairs */
  8.0, 0.5,
  10.0, 0.1,
  12.0, 0.0,
};

void init(void)
{
  GLUquadric *q = gluNewQuadric();
  GLfloat ambient[] = {0.3, 0.3, 0.2, 1.0};
  GLfloat diffuse[] = {0.8, 0.7, 0.8, 1.0};
  GLfloat specular[] = {0.5, 0.7, 0.8, 1.0};
  GLfloat lpos[] = {0.0, 10.0, −20.0, 0.0}; /* infinite light */
  GLfloat diff_mat[] = {0.1, 0.2, 0.5, 1.0};
  GLfloat amb_mat[] = {0.1, 0.2, 0.5, 1.0};
  GLfloat spec_mat[] = {0.9, 0.9, 0.9, 1.0};
  GLfloat shininess_mat[] = {0.8, 0.0};
  GLfloat amb_scene[] = {0.2, 0.2, 0.2, 1.0};
  GLfloat fog_color[] = {0.0, 0.0, 0.0, 1.0};

  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glFrustum(−4.0, 4.0, −4.0, 4.0, 4.0, 30.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  /* Setup lighting */

  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
  glLightfv(GL_LIGHT0, GL_POSITION, lpos);
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb_scene);

  glMaterialfv(GL_FRONT, GL_DIFFUSE, diff_mat);
  glMaterialfv(GL_FRONT, GL_AMBIENT, amb_mat);
  glMaterialfv(GL_FRONT, GL_SPECULAR, spec_mat);
  glMaterialfv(GL_FRONT, GL_SHININESS, shininess_mat);

  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);
  /* Setup fog function */

  glFogfv(GL_FOG_COLOR, fog_color);
  glFogf(GL_FOG_MODE, GL_FOG_FUNC_SGIS);
  glFogFuncSGIS(4, fogfunc);
  glEnable(GL_FOG);

  /* Setup scene */

  glTranslatef(0.0, 0.0, −6.0);
  glRotatef(60.0, 1.0, 0.0, 0.0);

  glNewList(1, GL_COMPILE);
  glPushMatrix();
  glTranslatef(2.0, 0.0, 0.0);
  glColor3f(1.0, 1.0, 1.0);
  gluSphere(q, 1.0, 40, 40);
  glTranslatef(−4.0, 0.0, 0.0);
  gluSphere(q, 1.0, 40, 40);
  glTranslatef(0.0, 0.0, −4.0);
  gluSphere(q, 1.0, 40, 40);
  glTranslatef(4.0, 0.0, 0.0);
  gluSphere(q, 1.0, 40, 40);
  glTranslatef(0.0, 0.0, −4.0);
  gluSphere(q, 1.0, 40, 40);
  glTranslatef(−4.0, 0.0, 0.0);
  gluSphere(q, 1.0, 40, 40);
  glPopMatrix();
  glEndList();
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  (dofog) ? glEnable(GL_FOG) : glDisable(GL_FOG);
  glCallList(1);
  glutSwapBuffers();
}

void kbd(unsigned char key, int x, int y)
{
  switch (key) {
  case ‘f’:                                       /* toggle fog ena
ble */
    dofog = 1 − dofog;
    glutPostRedisplay();
    break;

  case ESC:   /* quit!! */
    exit(0);
  }
}

main(int argc, char *argv[])
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
  glutInitWindowSize(width, height);
  glutCreateWindow("Fog Function");
  glutKeyboardFunc(kbd);
  glutDisplayFunc(display);

  init();
  glutMainLoop();
}
```

New Function

glFogFuncSGIS

SGIS_fog_offset—The Fog Offset Extension

The fog offset extension, SGIX_fog_offset, allows applications to make objects look brighter in a foggy environment.

When fog is enabled, it is equally applied to all objects in a scene. This can create unrealistic effects for objects that are especially bright (light sources like automobile headlights, runway landing lights, or florescent objects, for instance). To make such objects look brighter, fog offset may be subtracted from the eye distance before it is used for the fog calculation. This works appropriately because the closer an object is to the eye, the less obscured by fog it is.

To use fog with a fog offset, follow these steps:

1. Call *glEnable()* with the GL_FOG argument to enable fog.
2. Call *glFog*()* to choose the color and the equation that controls the density.

   The above two steps are explained in more detail in "Using Fog" on page 240 of the *OpenGL Programming Guide, Second Edition.*
3. Call *glEnable()* with argument GL_FOG_OFFSET_SGIX.
4. Call *glFog*()* with a *pname* of GL_FOG_OFFSET_VALUE_SGIX and four *params*. The first three parameters are point coordinates in the eye−space and the fourth parameter is an offset distance in the eye−space.

   The GL_FOG_OFFSET_VALUE_SGIX parameter specifies point coordinates in eye−space and offset amount toward the viewpoint. It is subtracted from the depth value (to make objects closer to the viewer) right before fog calculation. As a result, objects look less foggy. Note that these point coordinates are needed for OpenGL implementations that use z−based fog instead of eyespace distance. The computation of the offset in the z dimension is accurate only in the neighborhood of the specified point.

If the final distance is negative as a result of offset subtraction, it is clamped to 0. In the case of perspective projection, fog offset is properly calculated for the objects surrounding the given point. If objects are too far away from the given point, the fog offset value should be defined again. In the case of ortho projection, the fog offset value is correct for any object location.

5. Call *glDisable()* with argument GL_FOG_OFFSET_SGIX to disable fog offset.

SGIS_multisample—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 explores what happens when you use it. It discusses the following topics:

"Introduction to Multisampling"

"Using the Multisample Extension" and "Using Advanced Multisampling Options"

"How Multisampling Affects Different Primitives"

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, 16, or even more subsamples, depending on the platform.

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. This 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 of multisampling (see "Antialiasing" in Chapter 7, "Blending, Antialiasing, Fog, and Polygon Offset" of the *OpenGL Programming Guide).*

The multisampling extension lets you alternate multisample and smooth antialiasing during the rendering of a single scene, so it is possible to mix multisampled polygons with smooth lines and points. See "Multisampled Points" and "Multisampled Lines" 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 writable

using alpha values to feather−blend texture edges

using the accumulation buffer with multisampling

Figure 8−1shows 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").
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.

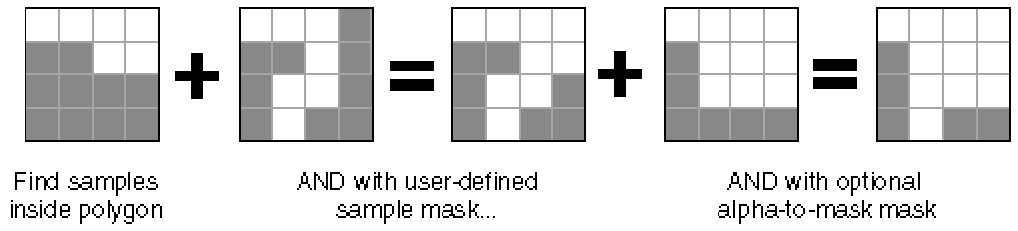

Find samples inside polygon
AND with user-defined sample mask...
AND with optional alpha-to-mask mask

Figure 8−1 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 is 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" 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 should 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," on page 394 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 are using multisampling, the value of the smoothing hint (GL_POINT_SMOOTH_HINT or GL_LINE_SMOOTH_HINT) is ignored. Because 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 and InfiniteReality systems, you achieve higher−quality multisampled points by setting point smooth hint set to 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 Note: in the next section for caveats on using multisampling with smooth points and lines.

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. 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 and InfiniteReality systems look better when drawn "smooth" than they do with multisampling.

Note: 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 will not necessarily be anti−aliased correctly if they intersect any of the lines or points.

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 stipple 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

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

SGIS_point_parameters—The Point Parameters Extension

The point parameter extension, SGIS_point_parameters can be used to render tiny light sources, commonly referred to as "light points." The extension is useful, for example, in an airport runway simulation. As the plane moves along the runway, the light markers grow larger as they approach.

Note: This extension is currently implemented on InfiniteReality systems only.

By default, a fixed point size is used to render all points, regardless of their distance from the eye point. Implementing the runway example or a similar scene would be difficult with this behavior. This extension is useful in two ways:

> It allows the size of a point to be affected by distance attenuation, that is, the point size decreases as the distance of the point from the eye increases.
>
> It increases the dynamic range of the raster brightness of points. In other words, the alpha component of a point may be decreased (and its transparency increased) as its area shrinks below a defined threshold. This is done by controlling the mapping from the point size to the raster point area and point transparency.

The new point size derivation method applies to all points, while the threshold applies to multisample points only. The extension makes this behavior available via the following constants:

> GL_POINT_SIZE_MIN_SGIS and GL_POINT_SIZE_MAX_SGIS define upper and lower bounds, respectively, for the derived point size.
>
> GL_POINT_FADE_THRESHOLD_SIZE_SGIS affects only multisample points. If the derived point size is larger than the threshold size defined by the GL_POINT_FADE_THRESHOLD_SIZE_SGIS parameter, the derived point size is used as the diameter of the rasterized point, and the alpha component is intact. Otherwise, the threshold size is set to be the diameter of the rasterized point, while the alpha component is modulated accordingly, to compensate for the larger area.

All parameters of the *glPointParameterfSGIS()* and *glPointParameterfvSGIS()* functions set various values applied to point rendering. The derived point size is defined to be the *size* provided as an argument to *glPointSize()* modulated with a distance attenuation factor.

Using the Point Parameters Extension

To use the point parameter extension, call *glPointParameter*SGIS()* with the following arguments:

> *pname* set to one of the legal arguments:

- GL_POINT_SIZE_MIN_SGIS
- GL_POINT_SIZE_MAX_SGIS
- GL_POINT_FADE_THRESHOLD_SIZE_SGIS (multisample points only)

param set to the value you want to set for the minimum size, maximum size, or threshold size of the point.

Note: If you are using the extension in multisample mode, you have to use smooth points to achieve the desired improvements:

```
glHint(GL_POINT_SMOOTH_HINT, GL_NICEST)
```

Point Parameters Example Code

A point parameters example program is available as part of the developer toolbox. It allows you to change the following attributes directly:

values of the distance attenuation coefficients (see "Point Parameters Background Information" and the point parameters specification)

fade threshold size

minimum and maximum point size

The following code fragment illustrates how to change the fade threshold.

Example 8−1 Point Parameters Example

```
GLvoid
decFadeSize( GLvoid )
{
#ifdef  GL_SGIS_point_parameters
    if (pointParameterSupported) {
        if ( fadeSize > 0 ) fadeSize −= 0.1;
        printf( "fadeSize = %4.2f\n", fadeSize );
        glPointParameterfSGIS( GL_POINT_FADE_THRESHOLD_SIZE_SGIS, fa
deSize );
        glutPostRedisplay();
    } else {
        fprintf( stderr,
                   "GL_SGIS_point_parameters not supported
                    on this machine\n");
    }
#else
    fprintf( stderr,
               "GL_SGIS_point_parameters not supported
                on this machine\n");
#endif
```

Minimum and maximum point size and other elements can also be changed; see the complete example program in the Developer Toolbox.

Point Parameters Background Information

The raster brightness of a point is a function of the point area, point color, and point transparency, and the response of the display's electron gun and phosphor. The point area and the point transparency are derived from the point size, currently provided with the *size* parameter of *glPointSize()*.

This extension defines a derived point size to be closely related to point brightness. The brightness of a point is given by the following equation:

```
                1
 dist_atten(d) = -------------------
                a + b * d + c * d^2
 brightness(Pe) = Brightness * dist_atten(|Pe|)
```

Pe is the point in eye coordinates, and *Brightness* is some initial value proportional to the square of the size provided with *glPointSize()*. The raster brightness is simplified to be a function of the rasterized point area and point transparency:

```
area(Pe) = brightness (Pe) if brightness(Pe) >= Threshold_Area
area(Pe) = Theshold_Area   otherwise

 factor(Pe) = brightness(Pe)/Threshold_Area

 alpha(Pe) = Alpha * factor(Pe)
```

`Alpha` comes with the point color (possibly modified by lighting). `Threshold_Area` is in area units. Thus, it is proportional to the square of the threshold provided by the programmer through this extension.

Note: For more background information, see the specification of the point parameters extension.

New Procedures and Functions

glPointParameterfSGIS, glPointParameterfvSGI

SGIX_reference_plane—The Reference Plane Extension

The reference plane extension, SGIX_reference_plane, allows applications to render a group of coplanar primitives without depth−buffering artifacts. This is accomplished by generating the depth values for all the primitives from a single reference plane rather than from the primitives themselves. Using the reference plane extension ensures that all primitives in the group have exactly the same depth value at any given sample point, no matter what imprecision may exist in the original specifications of the primitives or in the OpenGL coordinate transformation process.

Note: This extension is supported only on InfiniteReality systems.

The reference plane is defined by a four−component plane equation. When*glReferencePlaneSGIX()* is called, the equation is transformed by the adjoint of the composite matrix, the concatenation of model−view and projection matrices. The resulting clip−coordinate coefficients are transformed by the current viewport when the reference plane is enabled.

If the reference plane is enabled, a new z coordinate is generated for a fragment (xf, yf, zf). This z coordinate is generated from (xf, yf); it is given the same z value that the reference plane would have at (xf, yf).

Why Use the Reference Plane Extension?

Having such an auto−generated z coordinate is useful in situations where the application is dealing with a stack of primitives. For example, assume a runway for an airplane is represented by

- a permanent texture on the bottom
- a runway markings texture on top of the pavement
- light points representing runway lights on top of everything

All three layers are coplanar, yet it is important to stack them in the right order. Without a reference plane, the bottom layers may show through due to precision errors in the normal depth rasterization algorithm.

Using the Reference Plane Extension

If you know in advance that a set of graphic objects will be in the same plane, follow these steps:

1. Call *glEnable()* with argument GL_REFERENCE_PLANE_SGIX.
2. Call *glReferencePlane()* with the appropriate reference plane equation to establish the reference plane. The form of the reference plane equation is equivalent to that of an equation used by *glClipplane()* (see page 137 of the *OpenGL Programming Guide, Second Edition*).
3. Draw coplanar geometry that shares this reference plane.
4. Call *glDisable()* with argument GL_REFERENCE_PLANE_SGIX.

New Function

glReferencePlaneSGIX

SGIX_shadow, SGIX_depth_texture, and SGIX_shadow_ambient—The Shadow Extensions

This section discusses three extensions that are currently used together to create shadows:

- The depth texture extension, SGIX_depth_texture, defines a new depth texture internal format. While this extension has other potential uses, it is currently used for shadows only.
- The shadow extension, SGIX_shadow, defines two operations that can be performed on texture values before they are passed to the filtering subsystem.
- The shadow ambient extension, SGIX_shadow_ambient, allows for a shadow that is not black but instead has a different brightness.

This section first explores the concepts behind using shadows in an OpenGL program. It then discusses how to use the extension in the following sections:

"Shadow Extension Overview"

"Creating the Shadow Map"

"Rendering the Application From the Normal Viewpoint"

Code fragments from an example program are used throughout this section.

Note: A complete example program, shadowmap.c, is available as part of the Developer’s Toolbox.

Shadow Extension Overview

The basic assumption used by the shadow extension is that an object is in shadow when something else is closer to the light source than that object is.

Using the shadow extensions to create shadows in an OpenGL scene consists of several conceptual steps:

1. The application has to check that both the depth texture extension and the shadow extension are supported.
2. The application creates a shadow map; an image of the depth buffer from the point of view of the light.

   The application renders the scene from the point of view of the light source and copies the resulting depth buffer to a texture with internal format GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16_SGIX, GL_DEPTH_COMPONENT24_SGIX, or GL_DEPTH_COMPONENT32_SGIX (the SGIX formats are part of the depth texture extension).
3. The application renders the scene from the normal viewpoint. In that process, it sets up texture coordinate generation and the texture coordinate matrix such that for each vertex, the r coordinate is equal to the distance from the vertex to the plane used to construct the shadow map.

   Projection depends on the type of light. Normally, a finite light (spot) is most appropriate (in that case, perspective projection is used). An infinite directional light may also give good results because it doesn’t require soft shadows.

   Note that diffuse lights give only soft shadows and are therefore not well suited, although texture filtering will result in some blurriness. Note that it is theoretically possible to do an ortho projection for directional infinite lights. The lack of soft shadowing is not visually correct but may be acceptable.
4. For this second rendering pass, the application then enables the texture parameter GL_TEXTURE_COMPARE_SGIX, which is part of the shadow extension and renders the scene once more. For each pixel, the distance from the light (which was generated by interpolating the r texture coordinate) is compared with the shadow map stored in texture memory. The results of the comparison show whether the pixel being textured is in shadow.
5. The application can then draw each pixel that passes the comparison with luminance 1.0 and each shadowed pixel with a luminance of zero, or use the shadow ambient extension to apply ambient light with a value between 0 and 1 (for example, 0.5).

Creating the Shadow Map

To create the shadow map, the application renders the scene with the light position as the viewpoint and saves the depth map into a texture image, as illustrated in the following code fragment:

```
static void
generate_shadow_map(void)
{
  int x, y;
  GLfloat log2 = log(2.0);

  x = 1 << ((int) (log((float) width) / log2));
  y = 1 << ((int) (log((float) height) / log2));
  glViewport(0, 0, x, y);
  render_light_view();

  /* Read in frame−buffer into a depth texture map */
glCopyTexImage2DEXT(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT16_SGIX,
      0, 0, x, y, 0);

glViewport(0, 0, width, height);
}
```

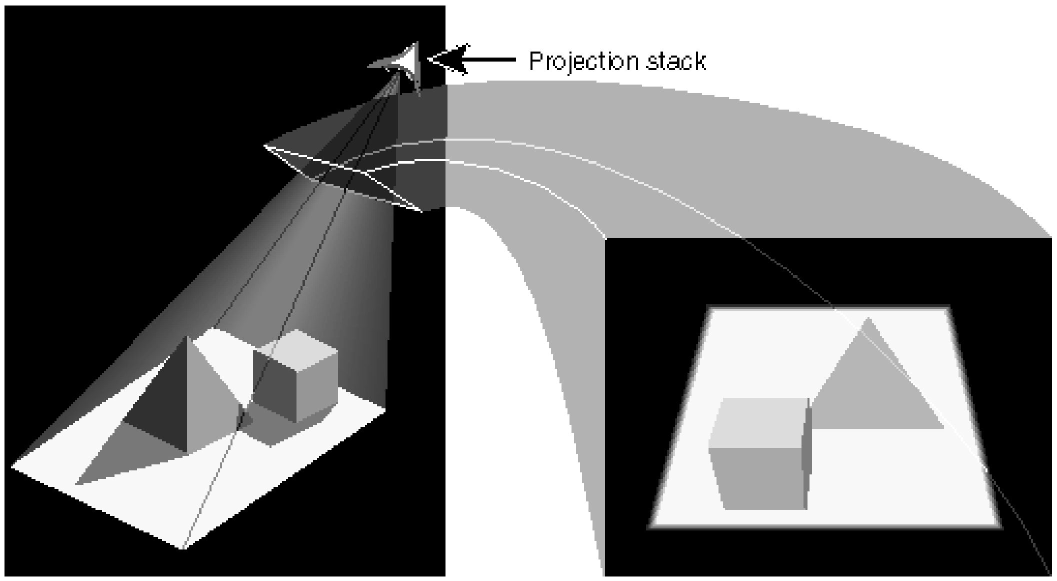


Figure 8−2 Rendering From the Light Source Point of View

Rendering the Application From the Normal Viewpoint

After generating the texture map, the application renders the scene from the normal viewpoint but with the purpose of generating comparison data. That is, use *glTexgen()* to generate texture coordinates that are identical to vertex coordinates. The texture matrix then transforms all pixel coordinates back to light coordinates. The depth−value is now available in the*r* texture coordinate.

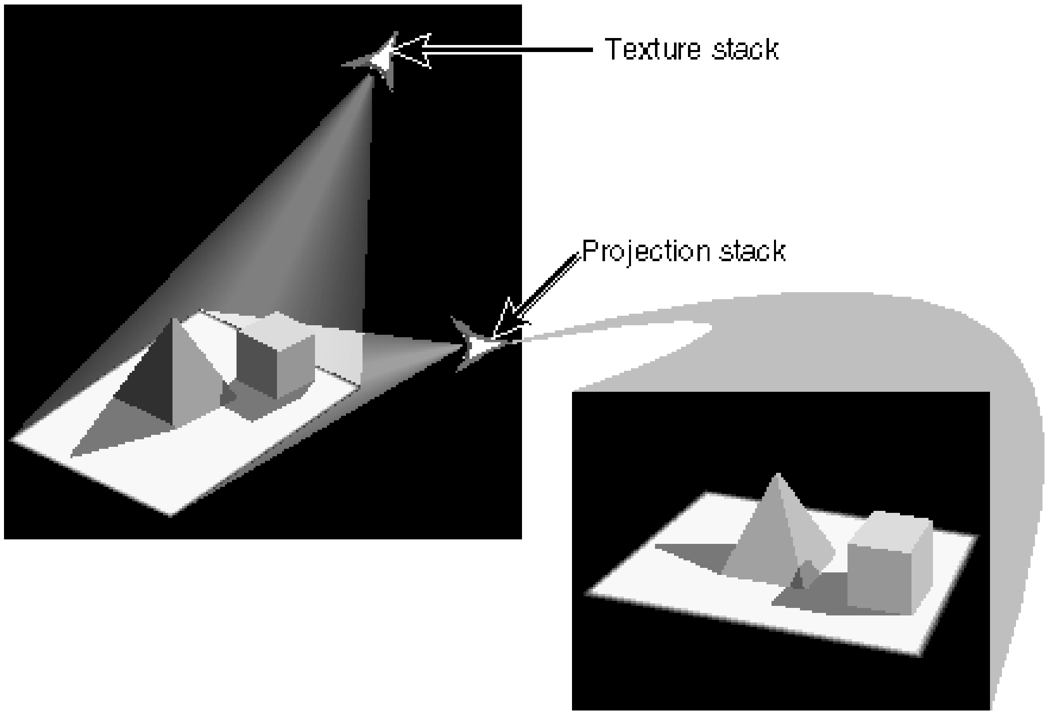


Figure 8−3 Rendering From Normal Viewpoint

During the second rendering pass, the r coordinate is interpolated over the primitive to give the distance from the light for every fragment. Then the texture hardware compares r for the fragment with the value from the texture. Based on this test, a value of 0 or 1 is sent to the texture filter. The application can render shadows as black, or use the shadow ambient extension discussed in the next section, to use a different luminance value.

Using the Shadow Ambient Extension

The shadow ambient extension allows applications to use reduced luminance instead of the color black for shadows. To achieve this, the extension makes it possible to return a value other than 0.0 by the SGIX_shadow operation in the case when the shadow test passes. With this extension any floating−point value in the range [0.0, 1.0] can be returned. This allows the (untextured) ambient lighting and direct shadowed lighting from a single light source to be computed in a single pass.

To use the extension, call *glTexParameter*()* with *pname* set to GL_SHADOW_AMBIENT_SGIX and *param* set to a floating−point value between 0.0 and 1.0. After the parameter is set, each pixel that extension is determined to be in shadow by the shadow extension has a luminance specified by this extension instead of a luminance of 0.0.

SGIX_sprite—The Sprite Extension

The sprite extension, SGIX_sprite, provides support for viewpoint−dependent alignment of geometry. In particular, geometry that rotates about a point or a specified axis is made to face the eye point at all times. Imagine, for example, an area covered with trees. As the user moves around in that area, it is important that the user always view the front of the tree. Because trees look similar from all sides, it makes sense to have each tree face the viewer (in fact, "look at" the viewer) at all times to create the illusion of a cylindrical object.

Note: This extension is currently available only on InfiniteReality systems.

Rendering sprite geometry requires applying a transformation to primitives before the current model view transformation is applied. This transformation matrix includes a rotation, which is computed based on

the current model view matrix

a translation that is specified explicitly (GL_ SPRITE_TRANSLATION_SGIX)

In effect, the model view matrix is perturbed only for the drawing of the next set of objects; it is not permanently perturbed.

This extension improves performance because the flat object you draw is much less complex than a true three−dimensional object would be. Platform−dependent implementations may need to ensure that the validation of the perturbed model view matrix has as small an overhead as possible. This is especially significant on systems with multiple geometry processors. Applications that intend to run on different systems benefit from verifying the actual performance improvement for each case.

Available Sprite Modes

Primitives are transformed by a rotation, depending on the sprite mode:

GL_SPRITE_AXIAL_SGIX: The front of the object is rotated about an *axis* so that it faces the eye as much as the axis constraint allows. This mode is used for rendering roughly cylindrical objects (such as trees) in a visual simulation. See Figure 8−4for an example.

GL_SPRITE_OBJECT_ALIGNED_SGIX: The front of the object is rotated about a *point* to face the eye. The remaining rotational degree of freedom is specified by aligning the top of the object with a specified axis in object coordinates. This mode is used for spherical symmetric objects (such as clouds) and for special effects such as explosions or smoke which must maintain an alignment in object coordinates for realism. See Figure 8−5for an example.

GL_SPRITE_EYE_ALIGNED_SGIX: The front of the object is rotated about a point to face the eye. The remaining rotational degree of freedom is specified by aligning the top of the object with a specified axis in eye coordinates. This is used for rendering sprites that must maintain an alignment on the screen, such as 3D annotations. See Figure 8−6for an example.

The axis of rotation or alignment, GL_SPRITE_AXIS_SGIX, can be in an arbitrary direction to support geocentric coordinate frames in which "up" is not along x, y, or z.

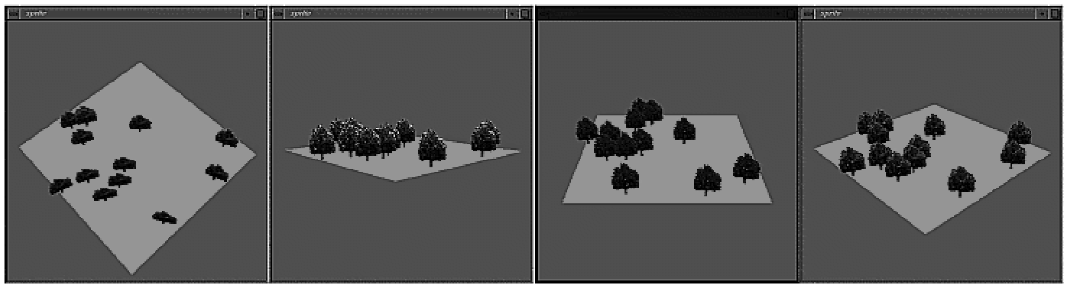


Figure 8−4 Sprites Viewed with Axial Sprite Mode

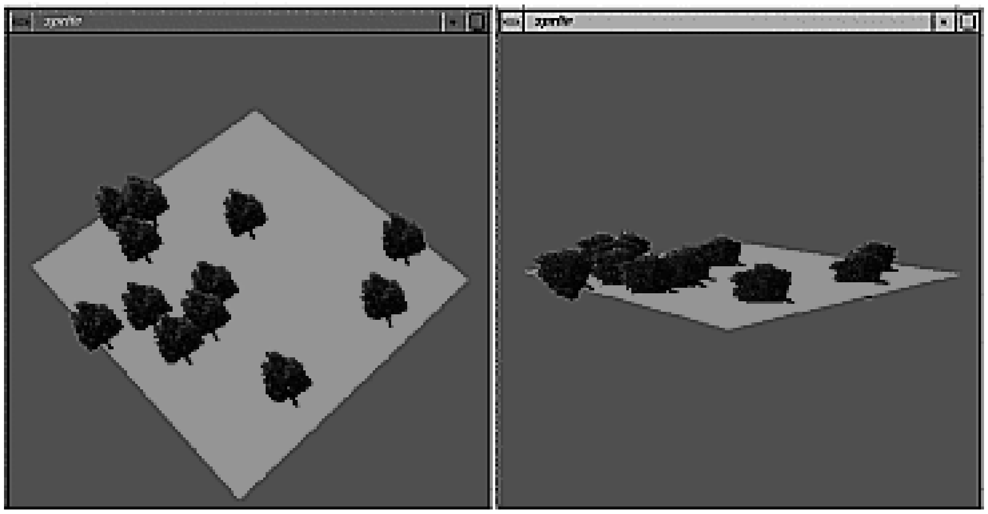


Figure 8−5 Sprites Viewed With Object Aligned Mode

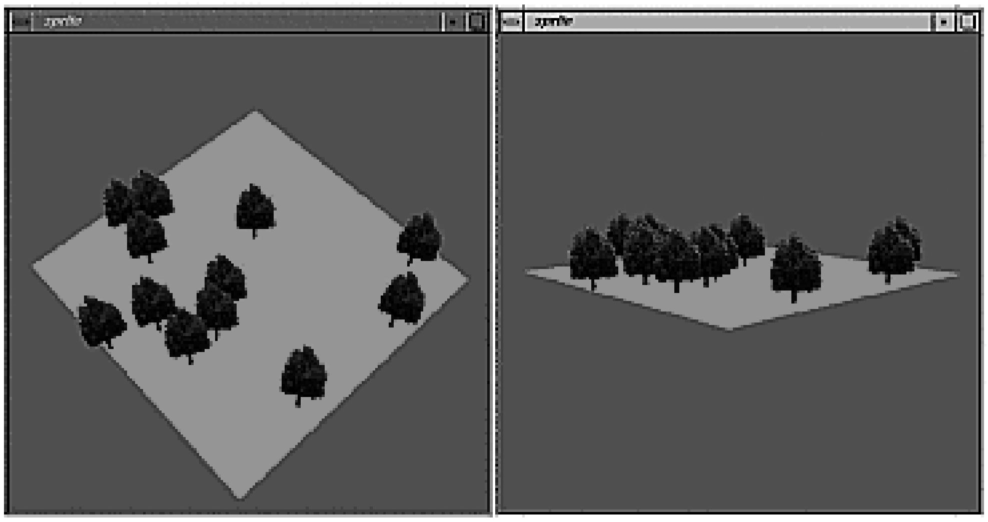


Figure 8−6 Sprites Viewed With Eye Aligned Mode

Note: The sprite extension specification discusses in more detail how the sprite transformation is computed. See "Extension Specifications" for more information.

Using the Sprite Extension

To render sprite geometry, an application applies a transformation to primitives before applying the current modelview matrix. The transformation is based on the current modelview matrix, the sprite rendering mode, and the constraints on sprite motion.

To use the sprite extension, follow these steps:

1. Enable sprite rendering by calling *glEnable()* with the argument GL_SPRITE_SGIX.
2. Call *glSpriteParameteriSGIX()* with one of the three possible modes: GL_SPRITE_AXIAL_SGIX, GL_SPRITE_OBJECT_ALIGNED_SGIX, or GL_SPRITE_EYE_ALIGNED_SGIX.
3. Specify the axis of rotation and the translation.

4. Draw the sprite geometry
5. Finally call *glDisable()* with the argument GL_SPRITE_SGIX and render the rest of the scene.

The following code fragment is from the *sprite.c* program in the OpenGL course "From the EXTensions to the SOLutions," which is available through the developer toolbox.

Example 8−2 Sprite Example Program

```
GLvoid
drawScene( GLvoid )
{
    int i,  slices = 8;

    glClear( GL_COLOR_BUFFER_BIT );

    drawObject();

    glEnable(GL_SPRITE_SGIX);
    glSpriteParameteriSGIX(GL_SPRITE_MODE_SGIX, GL_SPRITE_AXIAL_SGIX
);

/* axial mode (clipped geometry) */
    glPushMatrix();
    glTranslatef(.15, .0, .0);

    spriteAxis[0] = .2; spriteAxis[1] = .2; spriteAxis[2] = 1.0;
    glSpriteParameterfvSGIX(GL_SPRITE_AXIS_SGIX, spriteAxis);

    spriteTrans[0] = .2; spriteTrans[1] = .0; spriteTrans[2] = .0;
    glSpriteParameterfvSGIX(GL_SPRITE_TRANSLATION_SGIX, spriteTrans)
;
    drawObject();
    glPopMatrix();

/* axial mode (non−clipped geometry) */
    glPushMatrix();
    glTranslatef(.3, .1, .0);

    spriteAxis[0] = .2; spriteAxis[1] = .2; spriteAxis[2] = 0.5;
    glSpriteParameterfvSGIX(GL_SPRITE_AXIS_SGIX, spriteAxis);

    spriteTrans[0] = .2; spriteTrans[1] = .2; spriteTrans[2] = .0;
    glSpriteParameterfvSGIX(GL_SPRITE_TRANSLATION_SGIX, spriteTrans)
;

    drawObject();
    glPopMatrix();
/* object mode */
    glSpriteParameteriSGIX(GL_SPRITE_MODE_SGIX, GL_SPRITE_OBJECT_ALI
GNED_SGIX);

    glPushMatrix();
    glTranslatef(.0, .12, .0);

    spriteAxis[0] = .8; spriteAxis[1] = .5; spriteAxis[2] = 1.0;
    glSpriteParameterfvSGIX(GL_SPRITE_AXIS_SGIX, spriteAxis);

    spriteTrans[0] = .0; spriteTrans[1] = .3; spriteTrans[2] = .0;
    glSpriteParameterfvSGIX(GL_SPRITE_TRANSLATION_SGIX, spriteTrans)
;

    drawObject();
    glPopMatrix();


/* eye mode */
    glSpriteParameteriSGIX(GL_SPRITE_MODE_SGIX, GL_SPRITE_EYE_ALIGNE
D_SGIX);
    glPushMatrix();
    glTranslatef(.15, .25, .0);
    spriteAxis[0] = .0; spriteAxis[1] = 1.0; spriteAxis[2] = 1.0;
    glSpriteParameterfvSGIX(GL_SPRITE_AXIS_SGIX, spriteAxis);

    spriteTrans[0] = .2; spriteTrans[1] = .2; spriteTrans[2] = .0;
    glSpriteParameterfvSGIX(GL_SPRITE_TRANSLATION_SGIX, spriteTrans)
;

    drawObject();
    glPopMatrix();

    glDisable(GL_SPRITE_SGIX);

    glutSwapBuffers();
    checkError("drawScene");
}
```

The program uses the different sprite modes depending on user input.

Sprite geometry is modeled in a canonical frame: +Z is the up vector. −Y is the front vector which is rotated to point towards the eye.

New Function

glSpriteParameterSGIX

Chapter 9
Imaging Extensions

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

Introduction to Imaging Extensions

This section discusses where extensions are in the OpenGL imaging pipeline; it also lists the commands that may be affected by one of the imaging extensions.

Where Extensions Are in the Imaging Pipeline

The OpenGL imaging pipeline is shown in the *OpenGL Programming Guide, Second Edition* in the illustration "Drawing Pixels with glDrawPixels*()" in Chapter 8, "Drawing Pixels, Bitmaps, Fonts, and Images." The *OpenGL Reference Manual, Second Edition* also includes two overview illustrations and a detailed fold−out illustration in the back of the book.

Figure 9−1is a high−level illustration of pixel paths.

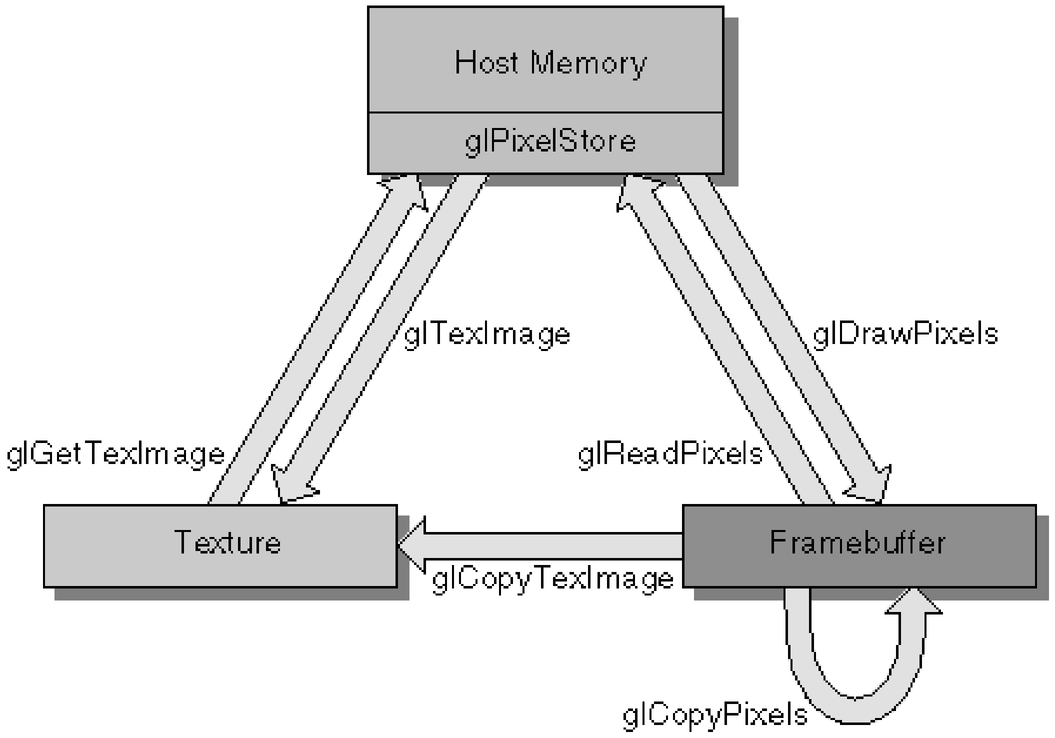


Figure 9−1 OpenGL Pixel Paths

The OpenGL pixel paths move rectangles of pixels between host memory, textures, and the framebuffer. Pixel store operations are applied to pixels as they move in and out of host memory. Operations defined by the *glPixelTransfer()* command, and other operations in the pixel transfer pipeline, apply to all paths between host memory, textures, and the framebuffer.

Pixel Transfer Paths

Certain pipeline elements, such as convolution filters and color tables are used during pixel transfer to modify pixels on their way to and from user memory, the framebuffer, and textures. The set of pixel paths used to initialize these pipeline elements is diagrammed in Figure 9−2 The pixel transfer pipeline is not applied to any of these paths.

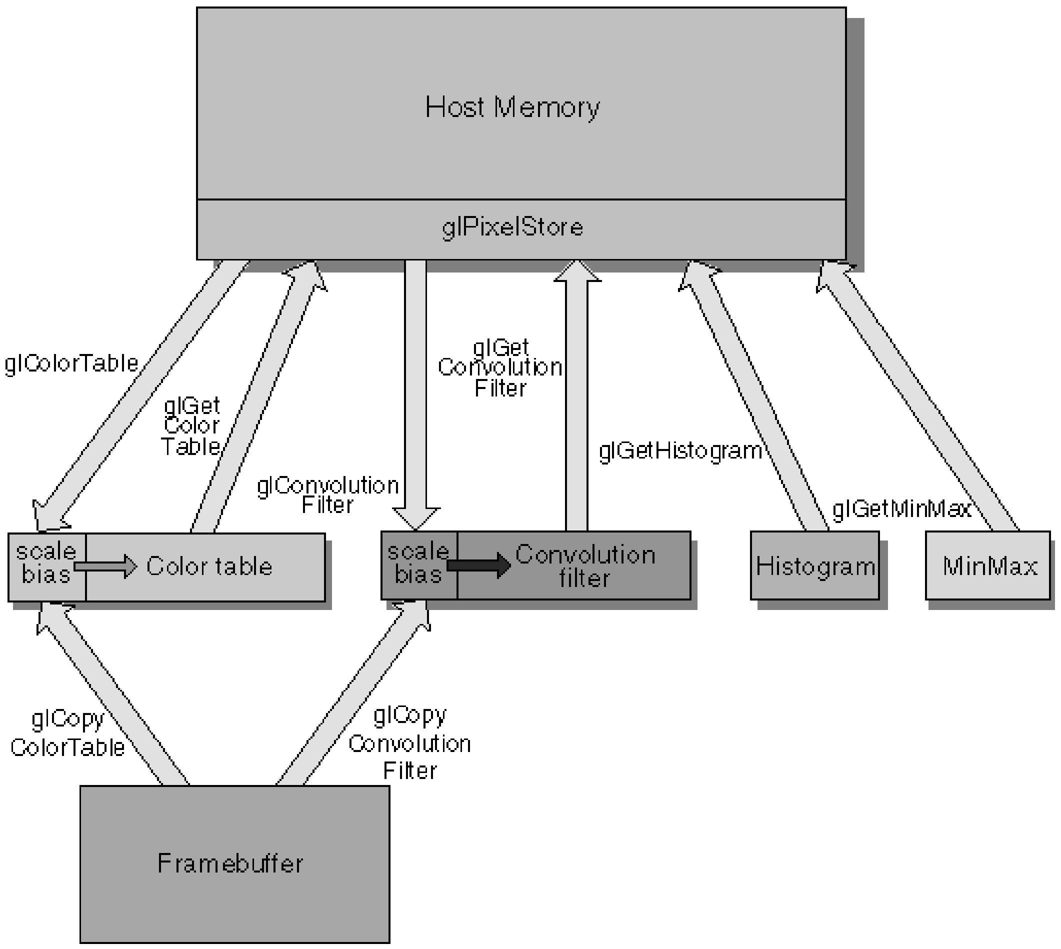


Figure 9−2 Extensions that Modify Pixels During Transfer

Convolution, Histogram, and Color Table in the Pipeline

Figure 9−3shows the same path with an emphasis on the position of each extension in the imaging pipeline itself. After the scale and bias operations and after the shift and offset operations, color conversion (LUT in Figure 9−3below) takes place with a lookup table. After that, the extension modules may be applied. Note how the color table extension can be applied at different locations in the pipeline. Unless the histogram or minmax extensions were called to collect information only, pixel processing continues, as shown in the *OpenGL Programming Guide*.

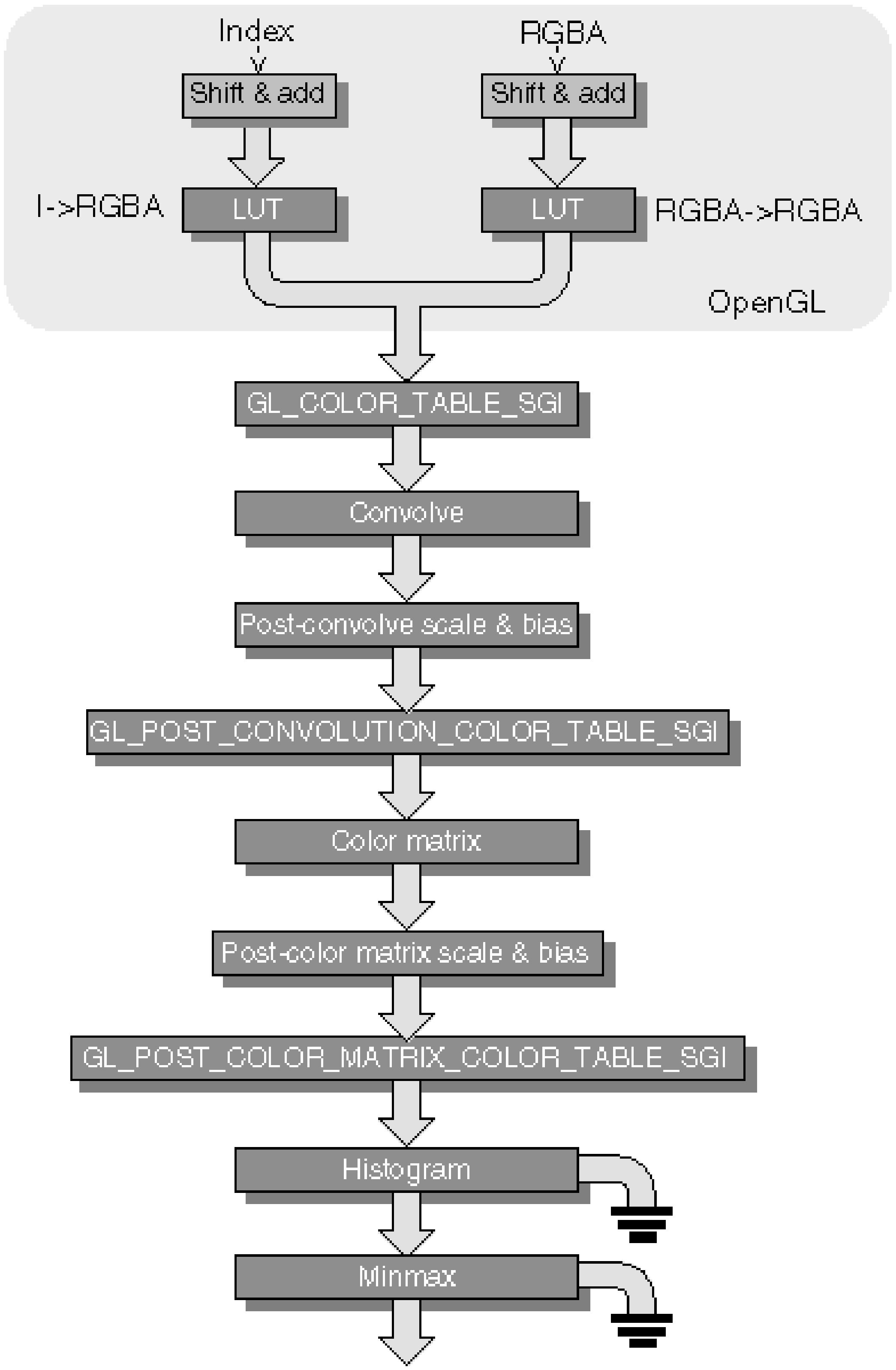

Index
RGBA
Shift & add
Shift & add
I->RGBA
LUT
LUT
RGBA->RGBA
OpenGL
GL_COLOR_TABLE_SGI
Convolve
Post-convolve scale & bias
GL_POST_CONVOLUTION_COLOR_TABLE_SGI
Color matrix
Post-color matrix scale & bias
GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI
Histogram
Minmax

Figure 9−3 Convolution, Histogram, and Color Table in the Pipeline

Interlacing and Pixel Texture in the Pipeline

Figure 9−4shows where interlacing (see "SGIX_interlace—The Interlace Extension") and pixel texture (see "SGIX_pixel_texture—The Pixel Texture Extension") are applied in the pixel pipeline.

The steps after interlacing are shown in more detail than the ones before to allow the diagram to include pixel texture.

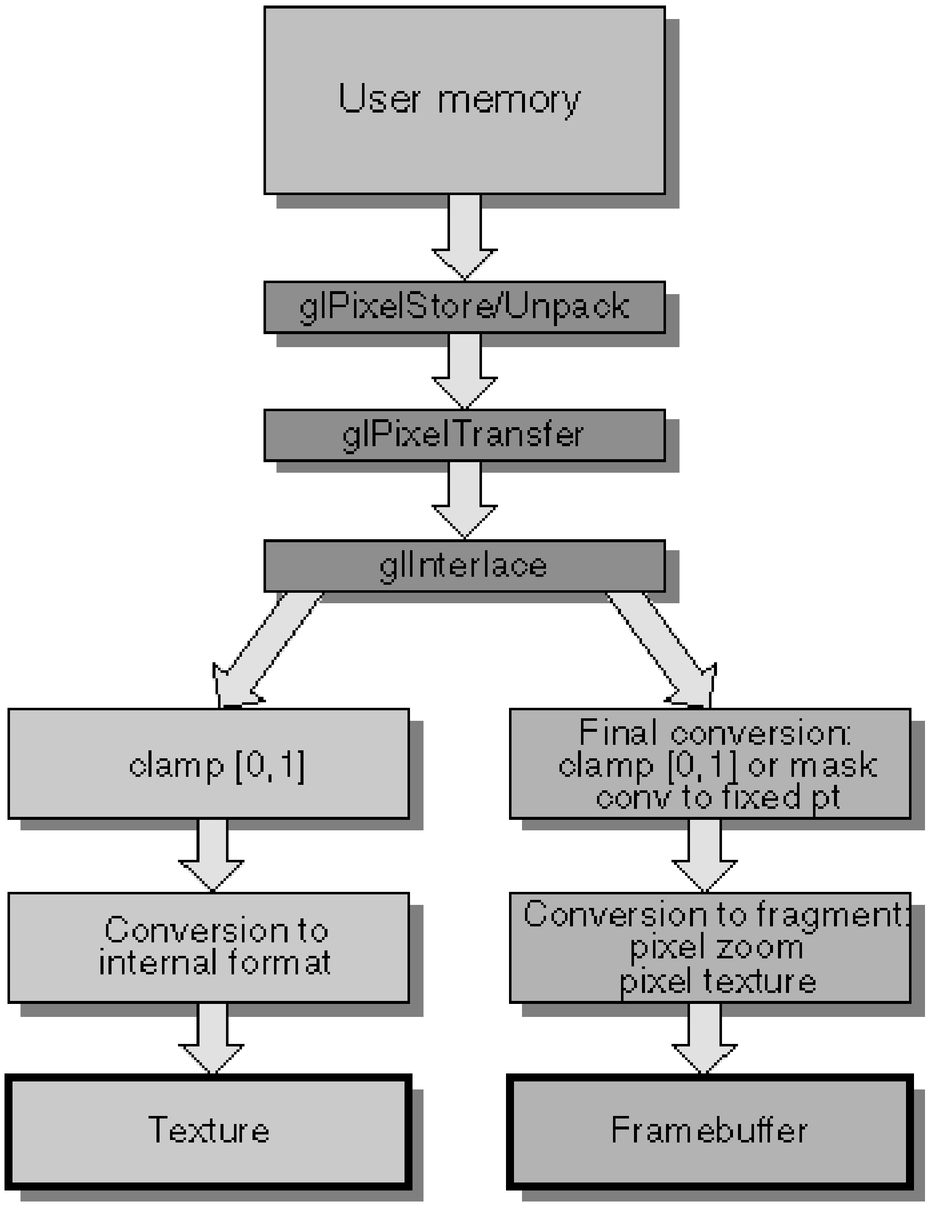

User memory
glPixelStore/Unpack
glPixelTransfer
glInterlace
clamp [0,1]
Final conversion: clamp [0,1] or mask conv to fixed pt
Conversion to internal format
Conversion to fragment: pixel zoom pixel texture
Texture
Framebuffer

Figure 9−4 Interlacing and Pixel Texture in the Pixel Pipeline

Merging the Geometry and Pixel Pipeline

The convert−to−fragment stage of geometry rasterization and of the pixel pipeline each produce fragments. The fragments are processed by a shared per−fragment pipeline that begins with applying the texture to the fragment color.

Because the pixel pipeline shares the per−fragment processing with the geometry pipeline, the fragments it produces must be identical to the ones produced by the geometry pipeline. The parts of the fragment that are not derived from pixel groups are filled with the associated values in the current raster position.

Pixel Pipeline Conversion to Fragments

A fragment consists of x and y window coordinates and its associated color value, depth value, and texture coordinates. The pixel groups processed by the pixel pipeline do not produce all the fragment's associated data, so the parts that are not produced from the pixel group are taken from the raster position. This combination of information allows the pixel pipeline to pass a complete fragment into the per fragment operations shared with the geometry pipeline, as shown in Figure 9−5

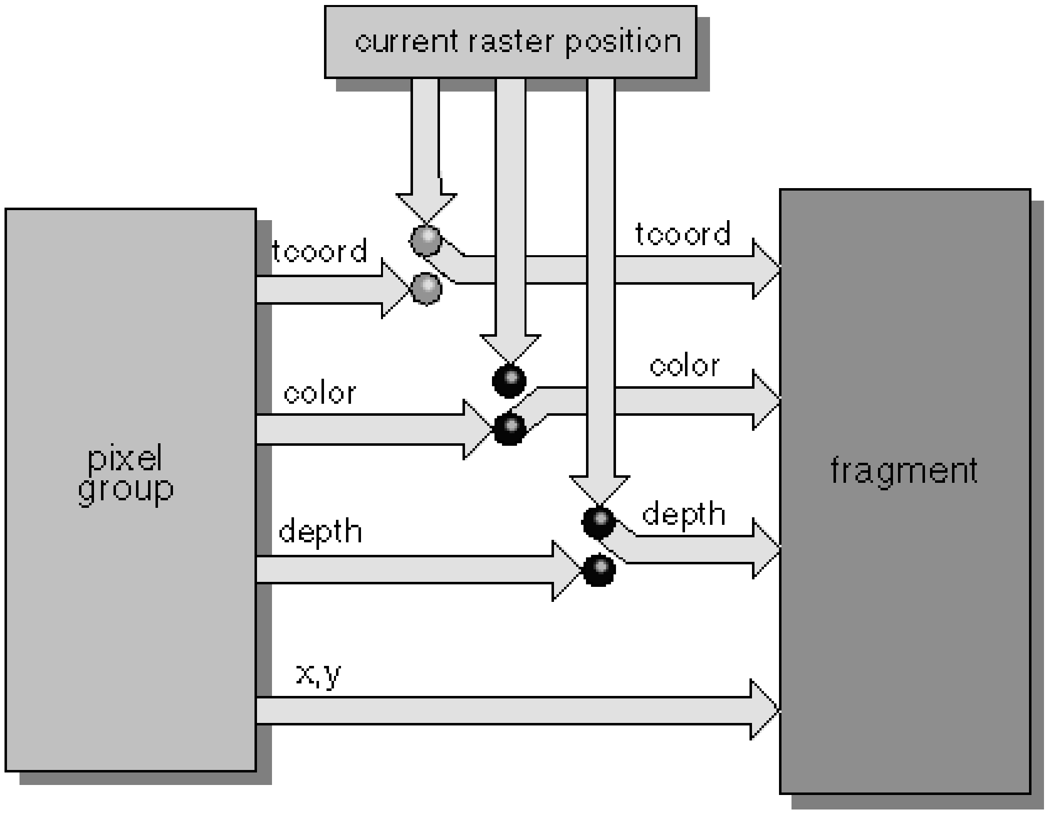


Figure 9−5 Conversion to Fragments

For example, if the pixel group is producing the color part of the fragment, the texture coordinates and depth value come from the current raster position. If the pixel group is producing the depth part of the fragment, the texture coordinates and color come from the current raster position.

The pixel texture extension (see "SGIX_pixel_texture—The Pixel Texture Extension") introduces the switch highlighted in blue, which provides a way to retrieve the fragment's texture coordinates from the pixel group. The pixel texture extension also allows developers to specify whether the color should come from the pixel group or the current raster position.

Functions Affected by Imaging Extensions

Imaging extensions affect all functions that are affected by the pixel transfer modes (see Chapter 8, "Drawing Pixels, Bitmaps, Fonts, and Images," of the *OpenGL Programming Guide).* In general, commands affected are

all commands that draw and copy pixels or define texture images

all commands that read pixels or textures back to host memory

EXT_abgr—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 some of the older 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 imag
e
 */

{
    unsigned char *image;

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

EXT_convolution—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 9−6shows 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 9−6 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, for example:

   ```
   glEnable(GL_CONVOLUTION_2D_EXT)
   ```

4. Perform pixel operations (for example pixel drawing or texture image definition).

   Convolution happens as the pixel operations are executed.

Retrieving Convolution State Parameters

If necessary, you can 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 typically 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()*, which has the following prototype:

```
void glSeparableFilter2DEXT( GLenum target,GLenum internalformat,GLsizei w
idth,
                             GLsizei height,GLenum format,GLenum type,
                             const GLvoid *row,const GLvoid *column )
```

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, 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 are 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, glConvolutionParameterEXT

EXT_histogram—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 9−7illustrates 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 times the color component falls within each bin. Assuming that 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.

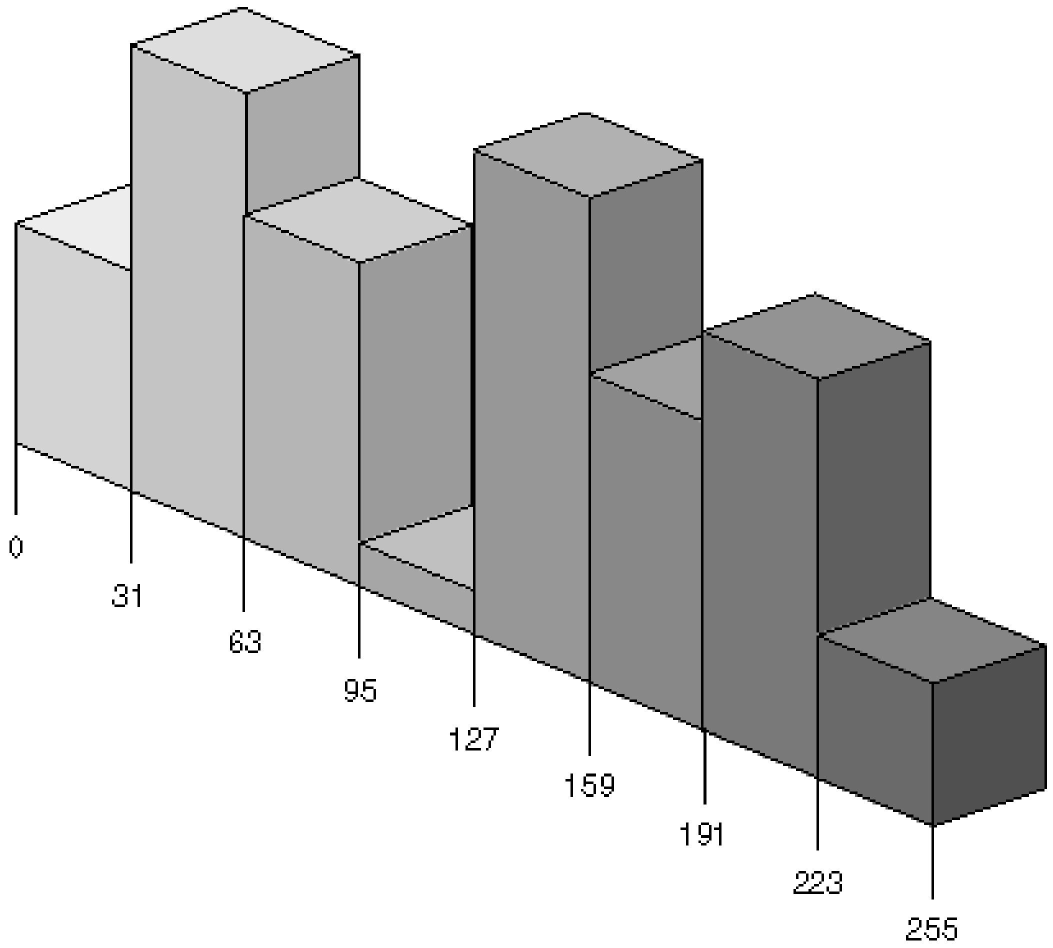


Figure 9−7 How the Histogram Extension Collects Information

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 histogram entries.

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).

1. Enable histogramming by calling

```
glEnable(GL_HISTOGRAM_EXT)
```

2. Perform the pixel operations for which you want to collect information (drawing, reading, copying pixels, or loading a texture). Only one operation is sufficient.

   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 `round((width−1)*c)` is incremented by 1.

3. 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, for example, GL_GREEN or GL_BLUE counters may not be returned if format is GL_RED and internal format is GL_RGB.

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 1D 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 operation, 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 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 the one explained in the section

"Texture Proxy" on page 330 of the *OpenGL Programming Guide, Second Edition.*

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

glGetHistogramEXT, glGetHistogramParameterEXT, glGetMinmaxEXT, glGetMinmaxParameterEXT, glHistogramEXT, glMinmaxEXT, glResetHistogramEXT, glResetMinmaxEXT

EXT_packed_pixels—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 have two potential benefits:

Save bandwidth. Packed pixels may use less bandwidth than unpacked pixels to transfer them to and from the graphics hardware because the packed pixel types use fewer bytes per pixel.

Save processing time. If the packed pixel type matches the destination (texture or framebuffer) type, packed pixels save processing time.

In addition, some of the types defined by this extension match the internal texture formats, so less processing is required to transfer texture images to texture memory. Internal formats are part of OpenGL 1.1, they were available as part of the texture extension in OpenGL 1.0.

Using Packed Pixels

To use packed pixels, provide one of the types listed in Table 9−1as the *type* parameter to *glDrawPixels()*, *glReadPixels()*, and so on.

Table 9−1 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," on page 293 of the *OpenGL Programming Guide.*

Pixel Type Descriptions

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.

SGI_color_matrix—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 only 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.

The color matrix is always applied to all pixel transfers. To disable it, load the identity 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 follows 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

SGI_color_table—The Color Table Extension

The color table extension, SGI_color_table, defines a new RGBA−format color lookup mechanism. It doesn't replace the color lookup tables provided by the color maps discussed in the *OpenGL Programming Guide* but provides additional lookup capabilities.

Unlike pixel maps, the color table extension's download operations go though the *glPixelStore()* unpack operations, the way *glDrawPixels()* does.

When a color table is applied to pixels, OpenGL maps the pixel format to the color table format.

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 9−3), 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 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*()" on page 310 of the *OpenGL Programming Guide*).
3. The R, G, B, and A components of each pixel are 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 one−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*()" on page 310 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.

Lookup Tables in the Image Pipeline

The three lookup tables exist at different points in the image pipeline (see Figure 9−3):

GL_COLOR_TABLE_SGI is located immediately after index lookup or RGBA to RGBA mapping, and immediately before the convolution operation.

GL_POST_CONVOLUTION_COLOR_TABLE_SGI is located immediately after the convolution operation (including its scale and bias operations) and immediately before 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 before 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

SGIX_interlace—The Interlace Extension

The interlace extension, SGIX_interlace, provides a way to interlace rows of pixels when rasterizing pixel rectangles or loading texture images.Figure 9−4illustrates how the extension fits into the imaging pipeline

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, as shown in the following illustration:

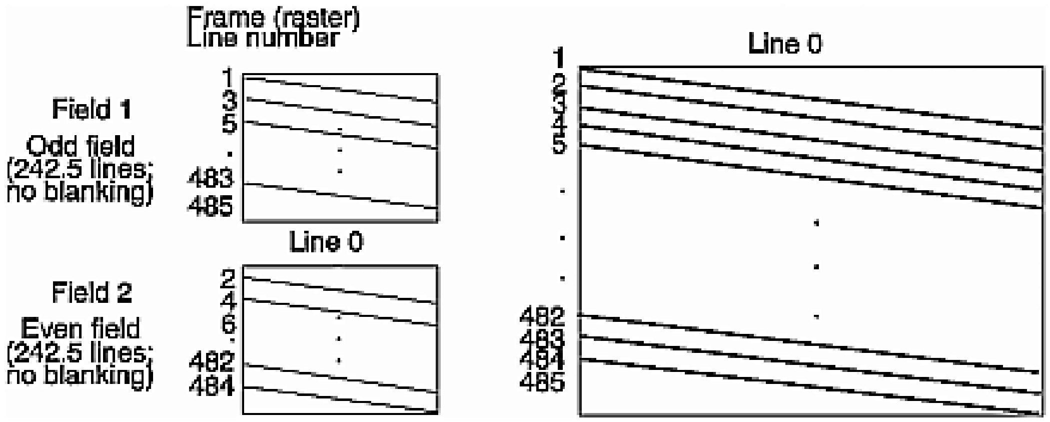


Figure 9−8 Interlaced Video (NTSC, Component 525)

When interlacing is enabled, all the groups that belong to a row *m* are treated as if they belonged to the row 2×μ. If the source image has a height of *h* rows, this effectively expands the height of the image to 2×η rows.

Applications that use the extension usually first copy the first set of rows, then the second set of rows, as explained in the following sections.

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`.

Using the Interlace Extension

One application of the interlace extension is to use it together with the copy texture extension: You can use *glCopyTexSubImage2D()* 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:

1. Call *glEnable()* or *glDisable()* with the *cap* parameter GL_INTERLACE_SGIX.
2. Set the current raster position to xr, yr:

   ```
   glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I0);
   ```
3. Copy pixels into texture memory (usually F0 is first).

   ```
   glCopyTexSubImage2D (GL_TEXTURE_2D, level, xoffset, yoffset, x, y
   ,
                        width, height)
   ```
4. Set raster position to (xr,yr+zoomy):

   ```
   glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I1);
   ```
5. Copy the pixels from the second field (usually F1 is next). For this call:

   ```
   y offset += yzoom
   y += geith to get to next field.
   ```

This process is equivalent to taking pixel rows (0,2,4,...) of I2 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);
```

SGIX_pixel_texture—The Pixel Texture Extension

The pixel texture extension, SGIX_pixel_texture, allows applications to use the color components of a pixel group as texture coordinates, effectively converting a color image into a texture coordinate image. Applications can use the system's texture−mapping capability as a multidimensional lookup table for images. Using larger textures will give you higher resolution, and the system will interpolate whenever the precision of the color values (texture coordinates) exceeds the size of the texture.

In effect, the extension supports multidimensional color lookups that can be used to implement accurate and fast color space conversions for images. Figure 9−4illustrates how the extension fits into the imaging pipeline.

Note: This extension is experimental and will change.

Texture mapping is usually used to map images onto geometry, and each pixel fragment that is generated by the rasterization of a triangle or line primitive derives its texture coordinates by interpolating the coordinates at the primitive's vertexes. Thus you don't have much direct control of the texture coordinates that go into a pixel fragment.

By contrast, the pixel texture extension gives applications direct control of texture coordinates on a per−pixel basis, instead of per−vertex as in regular texturing. If the extension is enabled, *glDrawPixels()* and *glCopyPixels()* work differently: For each pixel in the transfer, the color components are copied into the texture coordinates, as follows:

red becomes the *s* coordinate

green becomes the *t* coordinate

blue becomes the *r* coordinate

alpha becomes the *q* coordinate (fourth dimension)

To use the pixel texture extension, an application has to go through these steps:

1. Define and enable the texture you want to use as the lookup table (this texture doesn't have to be a 3D texture).

   ```
   glTexImage3DEXT(GL_TEXTURE_3D_EXT, args);
   glEnable(GL_TEXTURE_3D_EXT);
   ```

2. Enable pixel texture and begin processing images:

   ```
   glEnable(GL_PIXEL_TEX_GEN_SGIX);
   glDrawPixels(args);
   glDrawPixels(args)
   ...
   ...
   ```

Each subsequent call to *glDrawPixels()* uses the predefined texture as a lookup table and uses those colors when rendering to the screen. Figure 9−5illustrates how colors are introduced by the extension.

As in regular texture mapping, the texel found by mapping the texture coordinates and filtering the texture is blended with a pixel fragment, and the type of blend is controlled with the *glTexEnv()* command. In the case of pixel texture, the fragment color is derived from the pixel group; thus, using the GL_MODULATE blend mode, you could blend the texture lookup values (colors) with the original image colors. Alternatively, you could blend the texture values with a constant color set with the *glColor*()* commands. To do this, use this command:

```
void glPixelTexGenSGIX(GLenum mode);
```

The valid values of *mode* depend on the pixel group and the current raster color, which is the color associated with the current raster position:

GL_RGB—If *mode* is GL_RGB, the fragment red, green, and blue will be derived from the current raster color, set by the *glColor()* command. Fragment alpha is derived from the pixel group.

GL_RGBA—If mode is GL_RGBA, the fragment red, green, blue, and alpha will be derived from the current raster color.

GL_ALPHA—If mode is GL_ALPHA, the fragment alpha is derived from the current raster color, and red, green, and blue from the pixel group.

GL_NONE—If *mode* is GL_NONE, the fragment red, green, blue and alpha are derived from the pixel group.

Note: See "Platform Issues" for currently supported modes.

When using pixel texture, the format and type of the image do not have to match the internal format of the texture. This is a powerful feature; it means, for example, that an RGB image can look up a luminance result. Another interesting use is to have an RGB image look up an RGBA result, in effect adding alpha to the image in a complex way.

Platform Issues

At this date—IRIX 6.5—pixel texture has been implemented only on Indigo2 IMPACT and OCTANE graphics. The hardware capabilities for pixel texture were created before the OpenGL specifications for the extension were finalized, so only a subset of the full functionality has been implemented. Pixel texture can be enabled only with 3D and 4D textures.The only mode supported corresponds to calling `glPixelTexGenSGIX(GL_RGBA)`. The fragment color is limited to white only. Pixel texture can be enabled for *glDrawPixels()* only, support for *glCopyPixels()* will be provided in future releases.

When you use 4D textures with an RGBA image, the alpha value is used to derive Q, the fourth dimensional texture coordinate. Currently, the Q interpolation is limited to a default GL_NEAREST mode, regardless of the minfilter and magfilter settings.

When you work on Indigo IMPACT and OCTANE systems, you can use the Impact Pixel Texture extension, which allows applications to perform pixel texture operations with 4D textures, and accomplish the fourth interpolation with a two−pass operation, using the frame−buffer blend. For more information, see the impact_pixel_texture specification and the glPixelTexGenSGIX and glTexParameter reference pages.

Note: When working with mipmapped textures, the effective LOD value computed for each fragment is 0. The texture LOD and texture LOD bias extensions apply to pixel textures as well.

New Functions

glPixelTexGenSGIX

Chapter 10
Video Extensions

Chapter 6, "Resource Control Extensions," discusses a set of GLX extensions that can be used to control resources. This chapter provides information on a second set of GLX extension, extensions that support video functionality. You learn about

SGI_swap_control—The Swap Control Extension

The swap control extension, SGI_swap_control, allows applications to display frames at a regular rate, provided the time required to draw each frame can be bounded. The extension allows an application to set a minimum period for buffer swaps, counted in display retrace periods. (This is similar to the IRIS GL *swapinterval().*)

To set the buffer swap interval, call *glXSwapIntervalSGI()*, which has the following prototype:

```
int glXSwapIntervalSGI( int interval )
```

Specify 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. The new swap interval takes effect on the first execution of *glXSwapBuffers()* after the execution of *glXSwapIntervalSGI().*

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

SGI_video_sync—The Video Synchronization Extension

The video synchronization extension, SGI_video_sync, allows an application to synchronize drawing with the vertical retrace of a monitor or, more generically, to the boundary between to video frames. (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. This enables and application to synchronize itself with the start of a new video frame. 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.

Using the Video Sync Extension

To use the video sync extension, follow these steps:

1. Create a rendering context and make it current.
2. Call *glXGetVideoSyncSGI()* to obtain the value of the vertical retrace counter.
3. Call *glXWaitVideoSyncSGI()* to put the current process to sleep until the specified retrace counter:

   `int glXWaitVideoSyncSGI( int` *divisor*`, int` *remainder*`, unsigned int *`*count* `)`

   where

   glXWaitVideoSyncSGI() puts the calling process to sleep until the value of the vertical retrace counter (*count*) modulo divisor equals *remainder.*

   count is a pointer to the variable that receives the value of the vertical retrace counter when the calling process wakes up.

New Functions

glXGetVideoSyncSGI, glXWaitVideoSyncSGI

SGIX_swap_barrier—The Swap Barrier Extension

The swap barrier extension, SGIX_swap_barrier, allows applications to synchronize the buffer swaps of different swap groups. For information on swap groups, see "SGIX_swap_group—The Swap Group Extension".

Why Use the Swap Barrier Extension?

The swap barrier extension is useful for synchronizing buffer swaps of different swap groups, that is, on different machines.

For example, two Onyx InfiniteReality systems may be working together to generate a single visual experience. The first Onyx system may be generating an "out the window view" while the second Onyx system may be generating a sensor display. The swap group extension would work well if the two InfiniteReality graphics pipelines were in the same system, but a swap group can not span two Onyx systems. Even though the two displays are driven by independent systems, you still want the swaps to be synchronized.

The swap barrier solution requires the user to connect a physical coaxial cable to the "Swap Ready" port of each InfiniteReality pipeline. The multiple pipelines should also be genlocked together (synchronizing their video refresh rates). Genlocking a system means synchronizing it with another video signal serving as a master timing source.

The OpenGL swap barrier functionality requires special hardware support and is currently supported only on InfiniteReality graphics.

Note that most users of the swap barrier extension will likely use the extension through the IRIS Performer API and not call the OpenGL GLX extension directly.

Using the Swap Barrier Extension

A swap group is bound to a *swap_barrier*. The buffer swaps of each swap group using that barrier will wait until every swap group using that barrier is ready to swap (where readiness is defined in "Buffer Swap Conditions"). All buffer swaps of all groups using that barrier will take place concurrently when every group is ready.

The set of swap groups using the swap barrier include not only all swap groups on the calling application's system, but also any swap groups set up by other systems that have been cabled together by their graphics pipeline "Swap Ready" ports. This extension extends the set of conditions that must be met before a buffer swap can take place.

Applications call *glXBindSwapBarriersSGIX()*, which has the following prototype:

```
void glXBindSwapBarrierSGIX(Display *dpy, GLXDrawable drawable, int ba
rrier)
```

glXBindSwapBarriersSGIX() binds the swap group that contains *drawable* to *barrier*. Subsequent buffer swaps for that group will be subject to this binding until the group is unbound from *barrier*. If *barrier* is zero, the group is unbound from its current barrier, if any.

To find out how many swap barriers a graphics pipeline (an X screen) supports, applications call *glXQueryMaxSwapbarriersSGIX()*, which has the following syntax:

```
Bool glXQueryMaxSwapBarriersSGIX (Display *dpy, int screen, int max)
```

glXQueryMaxSwapBarriersSGIX() returns in *max* the maximum number of barriers supported by an implementation on *screen*.

glXQueryMaxSwapBarriersSGIX() returns GL_TRUE if it success and GL_FALSE if it fails. If it fails, *max* is unchanged.

While the swap barrier extension has the capability to support multiple swap barriers per graphics pipeline, InfiniteReality (the only graphics hardware currently supporting the swap barrier extension) provides only one swap barrier.

Buffer Swap Conditions

Before a buffer swap can take place when a swap barrier is used, some new conditions must be satisfied. The conditions are defined in terms of when a drawable is ready to swap and when a group is ready to swap.

Any GLX drawable that is not a window is always ready.

When a window is unmapped, it is always ready.

When a window is mapped, it is ready when both of the following are true:

- A buffer swap command has been issued for it.
- Its swap interval has elapsed.

A group is ready when all windows in the group are ready.

Before a buffer swap for a window can take place, all of the following must be satisfied:

- The window is ready.
- If the window belongs to a group, the group is ready.
- If the window belongs to a group and that group is bound to a barrier, all groups using that barrier are ready.

Buffer swaps for all windows in a swap group will take place concurrently after the conditions are satisfied for every window in the group.

Buffer swaps for all groups using a barrier will take place concurrently after the conditions are satisfied for every window of every group using the barrier, if and only if the vertical retraces of the screens of all the groups are synchronized (genlocked). If they are not synchronized, there is no guarantee of concurrency between groups.

Both *glXBindSwapBarrierSGIX()* and *glXQueryMaxSwapBarrierSGIX()* are part of the X stream.

New Functions

glBindSwapBarrierSGIX, glQueryMaxSwapBarriersSGIX

SGIX_swap_group—The Swap Group Extension

The swap group extension, SGIX_swap_group, allows applications to synchronize the buffer swaps of a group of GLX drawables.The application creates a swap group and adds drawables to the swap group. After the group has been established, buffer swaps to members of the swap group will take place concurrently.

In effect, this extension extends the set of conditions that must be met before a buffer swap can take place.

Why Use the Swap Group Extension?

Synchronizing the swapping of multiple drawables ensures that buffer swaps among multiple windows (potentially on different screens) swap at exactly the same time.

Consider the following example:

```
render(left_window);
render(right_window);
glXSwapBuffers(left_window);
glXSwapBuffers(right_window);
```

The *left_window* and *right_window* are on two different screens (different monitors) but are meant to generate a single logical scene (split across the two screens). While the programmer intends for the two swaps to happen simultaneously, the two *glXSwapBuffers()* calls are distinct requests, and buffer swaps are tied to the monitor’s rate of vertical refresh. Most of the time, the two *glXSwapBuffers()* calls will swap both windows at the next monitor vertical refresh. But because the two *glXSwapBuffers()* calls are not atomic, it is possible that:

the first *glXSwapBuffers()* call may execute just before a vertical refresh, allowing *left_window* to swap immediately,

the second *glXSwapBuffers()* call is made after the vertical refresh, forcing *right_window* to wait a full vertical refresh (typically a 1/60th or1/72th of a second).

Someone watching the results in the two windows would very briefly see the new *left_window* contents, but alongside the old *right_window* contents. This "stutter" between the two window swaps is always annoying and at times simply unacceptable.

The swap group extension allows applications to "tie together" the swapping of multiple windows.

By joining *left_window* and *right_window* into a swap group, IRIX ensures that the windows swap together atomically. This could be done during initialization by calling

```
glXJoinSwapGroupSGIX(dpy, left_window, right_window);
```

Subsequent windows can also be added to the swap group. For example, if there was also a middle window, it could be added to the swap group by calling

```
glXJoinSwapGroupSGIX(dpy, middle_window, right_window);
```

Swap Group Details

The only routine added by the swap group extension is *glXJoinSwapGroupSGIX()*, which has following prototype:

```
void glXJoinSwapGroupSGIX(Display *dpy, GLXDrawable drawable,
                          GLXDrawable member)
```

Applications can call *glXJoinSwapGroupSGIX()* to add *drawable* to the swap group containing *member* as a member. If *drawable* is already a member of a different group, it is implicitly removed from that group first. If *member* is None, *drawable* is removed from the swap group that it belongs to, if any.

Applications can reference a swap group by naming any drawable in the group; there is no other way to refer to a group.

Before a buffer swap can take place, a set of conditions must be satisfied. Both the drawable and the group must be ready, satisfying the following conditions:

GLX drawables, except windows, are always ready to swap.

When a window is unmapped, it is always ready.

When a window is mapped, it is ready when bothof the following are true:

– A buffer swap command has been issued for it.

– Its swap interval has elapsed.

A group is ready if all windows in the group are ready.

glXJoinSwapGroupSGIX() is part of the X stream. Note that a swap group is limited to GLX drawables managed by a single X server. If you have to synchronize buffer swaps between monitors on different machines, you need the swap barrier extension (see "SGIX_swap_barrier—The Swap Barrier Extension").

New Function

glJoinSwapGroupSGIX

SGIX_video_resize—The Video Resize Extension

The video resize extension, SGIX_video_resize, is an extension to GLX that allows the frame buffer to be dynamically resized to the output resolution of the video channel when *glXSwapBuffers* is called for the window that is bound to the video channel. The video resize extension can also be used to *minify* (reduce in size) a frame buffer image for display on a video output channel (such as NTSC or PAL broadcast video). For example, a 1280 x 1024 computer−generated scene could be minified for output to the InfiniteReality NTSC/PAL encoder channel. InfiniteReality performs bilinear filtering of the minified channel for reasonable quality.

As a result, an application can draw into a smaller viewport and spend less time performing pixel fill operations. The reduced size viewport is then magnified up to the video output resolution using the SGIX_video_resize extension.

In addition to the magnify and minify resizing capabilities, the video resize extension allows 2D panning. By overrendering at swap rates and panning at video refresh rates, it is possible to perform video refresh (frame) synchronous updates.

Controlling When the Video Resize Update Occurs

Whether frame synchronous or swap synchronous update is used is set by calling *glXChannelRectSyncSGIX()*, which has the following prototype:

```
int glXChannelRectSyncSGIX (Display *dpy, int screen,int channel,
                            GLenum synctype);
```

The *synctype* parameter can be either GLX_SYNC_FRAME_SGIX or GLX_SYNC_SWAP_SGIX.

The extension can control fill−rate requirements for real−time visualization applications or to support a larger number of video output channels on a system with limited framebuffer memory.

Note: This extension is an SGIX (experimental) extension. The interface or other aspects of the extension may change. The extension is currently implemented only on InfiniteReality systems.

Using the Video Resize Extension

To use the video resize extensions, follow these steps:

1. Open the display and create a window.
2. Call *glXBindChannelToWindowSGIX()* to associate a channel with an X window so that when the X window is destroyed, the channel input area can revert to the default channel resolution.

   The other reason for this binding is that the bound channel updates only when a swap takes place on the associated X window (assuming swap sync updates—see "Controlling When the Video

Resize Update Occurs").

The function has the following prototype:

```
int glXBindChannelToWindowSGIX( Display *display, int screen,
                                int channel, Window window )
```

where

display specifies the connection to the X server.

screen specifies the screen of the X server.

channel specifies the video channel number.

window specifies the window that is to be bound to *channel*. Note that InfiniteReality supports multiple output channels (two or eight depending on the Display Generator board type). Each channel can be *independently* dynamically resized.

3. Call *glXQueryChannelDeltasSGIX()* to retrieve the precision constraints for any frame buffer area that is to be resized to match the video resolution. In effect, *glXQueryChannelDeltasSGIX()* returns the resolution at which one can place and size a video input area.

   The function has the following prototype:

```
int glXQueryChannelDeltasSGIX( Display *display, int screen, int chan
nel,
                               int *dx, int *dy, int *dw, int *dh )
```

   where

   display specifies the connection to the X server.

   screen specifies the screen of the X server.

   channel specifies the video channel number.

   dx, dy, dw, dh are precision deltas for the origin and size of the area specified by *glXChannelRectSGIX()*

4. Call *XSGIvcQueryChannelInfo()* (an interface to the Silicon Graphics X video control X extension) to determine the default size of the channel.

5. Open an X window, preferably with no borders.

6. Start a loop in which you perform the following activities:
   - Determine the area that will be drawn, based on performance requirements. If the application is fill limited, make the area smaller. You can make a rough estimate of the fill rate required for a frame by timing the actual rendering time in milliseconds. On InfiniteReality, the SGIX_ir_instrument1 OpenGL extension can be used to query the pipeline performance to better estimate the fill rate.
   - Call *glViewPort()*, providing the width and height, to set the OpenGL viewport (the rectangular region of the screen where the window is drawn). Base this viewport on the information returned by *glXQueryChannelDeltasSGIX()*.
   - Call *glXChannelRectSGIX()* to set the input video rectangle that will take effect the next swap or next frame (based on *glXChannelRectSyncSGIX()* setting.) The coordinates of the

input video rectangle are those of the viewport just set up for drawing. This function has the following prototype:

```
int glXChannelRectSGIX( Display *display, int screen,
                       int channel, Window window)
```

where

display specifies the connection to the X server

screen specifies the screen of the X server.

channel specifies the video channel number.

x, y, w, h are the origin and size of the area of the window that will be converted to the output resolution of the video channel. (x,y) is relative to the bottom left corner of the channel specified by the current video combination.

- Draw the scene.
- Call *glXSwapBuffers()* for the window in question.

Example

The following example, from the glxChannelRectSGIX reference page, illustrates how to use the extension.

Example 10−1 Video Resize Extension Example

```
XSGIvcChannelInfo   *pChanInfo = NULL;

... open display and screen ...
glXBindChannelToWindowSGIX( display,screen,channel,window );
glXQueryChannelDeltasSGIX( display,screen,channel, &dx,&dy,&dw,&dh )
;

XSGIvcQueryChannelInfo( display, screen, channel, &pChanInfo );

X = pChanInfo−>source.x;
Y = pChanInfo−>source.y;
W = pChanInfo−>source.width;
H = pChanInfo−>source.height;

... open an X window (preferably with no borders so will not get ...
... moved by window manager) at location X,Y,W,H (X coord system) ..
.

while( ... )
{
    ...determine area(width,height) that will be drawn based on...
    ...requirements. Make area smaller if application is fill limite
d..
```

```
    w =  width − ( width % dw );
    h =  height − ( height % dh );

    glViewport( 0,0,w,h );

    glXChannelRectSGIX( display,screen,channel, 0,0,w,h );

    ... draw scene ...

    glXSwapBuffers( display,window );
}
```

New Functions

glXBindChannelToWindowSGIX, glXChannelRectSGIX, glXChannelRectSyncSGIX, glXQueryChannelRectSGIX

SGIX_video_source—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, to texture memory, or to a DMPbuffer (see "SGIX_pbuffer—The Pixel Buffer Extension").

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 be used only as the *read* parameter to *glXMakeCurrentReadSGI()* to indicate that pixel data should be read from the specified video source instead of the framebuffer.

Note: 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 10−2demonstrates the video to graphics capability of the Sirius video board using OpenGL. Example 10−3is a code fragment for how to use the video source extension to load video into texture memory.

Example 10−2 Use of the Video Source Extension

```
/*
 * vidtogfx.c
 *  This VL program demonstrates the Sirius Video board video−>graph
ics
 *  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 fir
st
   * 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 10−3 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 11
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:

"GLU_EXT_NURBS_tessellator—The NURBS Tessellator Extension"

"GLU_EXT_object_space—The Object Space Tess Extension"

"SGIX_instruments—The Instruments Extension"

"SGIX_list_priority—The List Priority Extension"

GLU_EXT_NURBS_tessellator—The NURBS Tessellator Extension

The NURBS tessellator extension, GLU_EXT_nurbs_tessellator, is a GLU extension that allows applications to retrieve the results of a tessellation. The NURBS tessellator is similar to the GLU polygon tessellator; see "Polygon Tessellation," starting on page 410 of the *OpenGL Programming Guide, Second Edition.*

NURBS tessellation consists of OpenGL Begin, End, Color, Normal, Texture, and Vertex data. This feature is useful for applications that need to cache the primitives to use their own advanced shading model, or to accelerate frame rate or perform other computations on the tessellated surface or curve data.

Using the NURBS Tessellator Extension

To use the extension, follow these steps:

1. Define a set of callbacks for a NURBS object using this command:

   ```
   void gluNurbsCallback(GLUnurbsObj *nurbsObj, GLenum which,
                         void (*fn)());
   ```

   The parameter *which* can be either GLU_ERROR or a data parameter or nondata parameter, one of the following:

GLU_NURBS_BEGIN_EXT	GLU_NURBS_BEGIN_DATA_EXT
GLU_NURBS_VERTEX_EXT	GLU_NURBS_VERTEX_DATA_EXT
GLU_NORMAL_EXT	GLU_NORMAL_DATA_EXT
GLU_NURBS_COLOR_EXT	GLU_NURBS_COLOR_DATA_EXT
GLU_NURBS_TEXTURE_COORD_EXT	GLU_NURBS_TEXTURE_COORD_DATA _EXT
GLU_END_EXT	GLU_END_DATA_EXT

2. Call *gluNurbsProperty()* with a *property* parameter of GLU_NURBS_MODE_EXT and *value* parameter of GLU_NURBS_TESSELLATOR_EXT or GLU_NURBS_RENDERER_EXT.

   In rendering mode, the objects are converted or tessellated to a sequence of OpenGL primitives, such as evaluators and triangles, and sent to the OpenGL pipeline for rendering. In tessellation mode, objects are converted to a sequence of triangles and triangle strips and returned to the application through a callback interface for further processing. The decomposition algorithms used for rendering and for returning tessellations are not guaranteed to produce identical results.

3. Execute your OpenGL code to generate the NURBS curve or surface (see "A Simple NURBS Example" on page 455 of the *OpenGL Programming Guide*, *Second Edition*.)
4. During tessellation, your callback functions are called by OpenGL, with the tessellation information defining the NURBS curve or surface.

Callbacks Defined by the Extension

There are two forms of each callback defined by the extension: one with a pointer to application supplied data and one without. If both versions of a particular callback are specified, the callback with *userData* will be used. *userData* is a copy of the pointer that was specified at the last call to *gluNurbsCallbackDataEXT()*.

The callbacks have the following prototypes:

```
void begin(GLenum type);
void vertex(GLfloat *vertex);
void normal(GLfloat *normal);
void color(GLfloat *color);
void texCoord(GLfloat *texCoord);
void end(void);
void beginData(GLenum type, void* userData);
void vertexData(GLfloat *vertex, void* userData);
void normalData(GLfloat *normal, void* userData);
void colorData(GLfloat *color, void* userData);
void texCoordData(GLfloat *texCoord, void* userData);
void endData(void* userData);
void error(GLenum errno);
```

The first 12 callbacks allows applications to get primitives back from the NURBS tessellator when GLU_NURBS_MODE_EXT is set to GLU_NURBS_TESSELLATOR_EXT.

These callbacks are not made when GLU_NURBS_MODE_EXT is set to GLU_NURBS_RENDERER_EXT.

All callback functions can be set to NULL even when GLU_NURBS_MODE_EXT is set to GLU_NURBS_TESSELLATOR_EXT. When a callback function is set to NULL, this function will not be invoked and the related data, if any, will be lost.

Table 11−1provides additional information on each callback.

Table 11−1NURBS Tessellator Callbacks and Their Description

Callback	**Description**
GL_NURBS_BEGIN_EXT GL_NURBS_BEGIN_DATA_ EXT	Indicates the start of a primitive. *type* is one of GL_LINES, GL_LINE_STRIPS, GL_TRIANGLE_FAN, GL_TRIANGLE_STRIP, GL_TRIANGLES, or GL_QUAD_STRIP. The default *begin()* and *beginData()* callback functions are NULL.
GL_NURBS_VERTEX_EXT GL_NURBS_VERTEX_DATA_ EXT	Indicates a vertex of the primitive. The coordinates of the vertex are stored in the parameter *vertex*. All the generated vertices have dimension 3, that is, homogeneous coordinates have been transformed into affine coordinates. The default *vertex()* and *vertexData()* callback functions are NULL.
GL_NURBS_NORMAL_EXT GL_NURBS_NORMAL_DATA_	Is invoked as the vertex normal is generated. The components of the normal are stored in the parameter *normal*. In the case of a NURBS curve, the callback function is effective only when

EXT	the user provides a normal map (GLU_MAP1_NORMAL). In the case of a NURBS surface, if a normal map (GLU_MAP2_NORMAL) is provided, then the generated normal is computed from the normal map. If a normal map is not provided, then a surface normal is computed in a manner similar to that described for evaluators when GL_AUTO_NORMAL is enabled. The default *normal()* and *normalData()* callback functions are NULL.
GL_NURBS_COLOR_EXT GL_NURBS_COLOR_DATA_ EXT	Is invoked as the color of a vertex is generated. The components of the color are stored in the parameter *color*. This callback is effective only when the user provides a color map (GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4). *color* contains four components: R, G, B, or A.The default *color()* and *colorData()* callback functions are NULL.
GL_NURBS_TEXCOORD_EXT GL_NURBS_TEXCOORD_ DATA_EXT	Is invoked as the texture coordinates of a vertex are generated. These coordinates are stored in the parameter *tex_coord*. The number of texture coordinates can be 1, 2, 3, or 4 depending on which type of texture map is specified (GL_MAP*_TEXTURE_COORD_1, GL_MAP*_TEXTURE_COORD_2, GL_MAP*_TEXTURE_COORD_3, GL_MAP*_TEXTURE_COORD_4 where * can be either 1 or 2). If no texture map is specified, this callback function will not be called. The default *texCoord()* and *texCoordData()* callback functions are NULL.
GL_NURBS_END_EXT GL_NURBS_END_DATA_EXT	Is invoked at the end of a primitive. The default *end()* and *endData()* callback functions are NULL.
GL_NURBS_ERROR_EXT	Is invoked when a NURBS function detects an error condition. There are 37 errors specific to NURBS functions. They are named GLU_NURBS_ERROR1 through GLU_NURBS_ERROR37. Strings describing the meaning of these error codes can be retrieved with *gluErrorString()*.

GLU_EXT_object_space—The Object Space Tess Extension

The object space tess extension, GLU_EXT_object_space_tess, adds two object space tessellation methods for GLU nurbs surfaces. NURBS are discussed in the section "The GLU NURBS Interface" on page 455 of the *OpenGL Programming Guide, Second Edition*.

The existing tessellation methods GLU_PATH_LENGTH and GLU_PARAMETRIC_ERROR are view dependent because the error tolerance is measured in the screen space (in pixels). The extension provides corresponding object space tessellation methods that are view−independent in that the error tolerance measurement is in the object space.

GLU_SAMPLING_METHOD specifies how a NURBS surface should be tessellated. The *value* parameter may be set to one of GLU_PATH_LENGTH, GLU_PARAMETRIC_ERROR, GLU_DOMAIN_DISTANCE, GLU_OBJECT_PATH_LENGTH_EXT, or GLU_OBJECT_PARAMETRIC_ERROR_EXT.

To use the extension, call *gluNurbsProperty()* with an argument of GLU_OBJECT_PATH_LENGTH_EXT or GLU_OBJECT_PARAMETRIC_ERROR_EXT. Table 11−2contrasts the methods provided by the extension with the existing methods.

Table 11−2 Tessellation Methods

Method	**Description**
GLU_PATH_LENGTH	The surface is rendered so that the maximum length, in pixels, of edges of the tessellation polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE.
GLU_PARAMETRIC_ERROR	The surface is rendered in such a way that the value specified by GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in pixels, between the tessellation polygons and the surfaces they approximate.
GLU_DOMAIN_DISTANCE	Allows you to specify, in parametric coordinates, how many sample points per unit length are taken in u, v dimension.

GLU_OBJECT_PATH_LENGTH_ EXT	Similar to GLU_PATH_LENGTH except that it is view independent; that is, it specifies that the surface is rendered so that the maximum length, in object space, of edges of the tessellation polygons is no greater than what is specified by GLU_SAMPLING_TOLERANCE.
GLU_OBJECT_PARAMETRIC_ ERROR_EXT	Similar to GLU_PARAMETRIC_ERROR except that it is view independent; that is, it specifies that the surface is rendered in such a way that the value specified by GLU_PARAMETRIC_TOLERANCE describes the maximum distance, in object space, between the tessellation polygons and the surfaces they approximate.

The default value of GLU_SAMPLING_METHOD is GLU_PATH_LENGTH.

GLU_SAMPLING_TOLERANCE specifies the maximum distance, in pixels or in object space when the sampling method is set to GLU_PATH_LENGTH or GLU_OBJECT_PATH_LENGTH_EXT. The default value for GLU_SAMPLING_TOLERANCE is 50.0.

GLU_PARAMETRIC_TOLERANCE specifies the maximum distance, in pixels or in object space when the sampling method is set to GLU_PARAMETRIC_ERROR or GLU_OBJECT_PARAMETRIC_ERROR_EXT. The default value for GLU_PARAMETRIC_TOLERANCE is 0.5.

SGIX_list_priority—The List Priority Extension

The list priority extension, SGIX_list_priority, provides a mechanism for specifying the relative importance of display lists. This information can be used by an OpenGL implementation to guide the placement of display list data in a storage hierarchy, that is, lists that have higher priority reside in "faster" memory and are less likely to be swapped out to make space for other lists.

Using the List Priority Extension

To guide the OpenGL implementation in determining which display lists should be favored for fast executions, applications call *glListParameter*SGIX()*, which has the following prototype:

```
glListParameterfSGIX(uint list, enum pname, float params)
```

where

list is set to the display list.

pname is set to GL_LIST_PRIORITY_SGIX.

params is set to the priority value.

The priority value is clamped to the range [0.0, 1.0] before it is assigned. Zero indicates the lowest priority, and hence the least likelihood of optimal execution. One indicates the highest priority, and hence the greatest likelihood of optimal execution.

Attempts to prioritize nonlists are silently ignored. Attempts to prioritize list 0 generates a GL_INVALID_VALUE error.

To query the priority of a list, call *glGetListParameterfvSGIX()*, which has the following prototype:

```
glGetListParameterivSGIX(uint list, enum pname, int *params)
```

where:

list is set to the list.

pname is set to GL_LIST_PRIORITY_SGIX.

If *list* is not defined, then the value returned is undefined.

Note: On InfiniteReality systems, it makes sense to give higher priority to those display lists that are changed frequently.

New Functions

glListParameterSGIX, glGetListParameterSGIX

SGIX_instruments—The Instruments Extension

The instruments extension, SGIX_instruments, allows applications to gather and return performance measurements from within the graphics pipeline by adding instrumentation.

About SGIX_instruments

There are two reasons for using the instruments extension:

Load monitoring. If you know that the pipeline is stalled or struggling to process the amount of data passed to it so far, you can take appropriate steps, such as these:

– Reduce the level of detail of the remaining objects in the current frame or the next frame.

– Adjust the framebuffer resolution for the next frame if video resize capability is available.

Tuning. The instrumentation may give you tuning information; for example, it may provide information on how many triangles were culled or clipped before being rasterized.

Load monitoring requires that the instrumentation and the access of the measurements be efficient, otherwise the instrumentation itself will reduce performance more than any load−management scheme could hope to offset. Tuning does not have the same requirements.

The instruments extension provides a call to set up a measurements return buffer, similar to the feedback buffer. However, unlike feedback and selection (see *glSelectBuffer()* and *glFeedbackBuffer()*), the instruments extension provides commands that allow measurements to be delivered asynchronously, so that the graphics pipeline need not be stalled while measurements are returned to the client.

Note that the extension provides an instrumentation framework, but no instruments. The set of available instruments varies between OpenGL implementations, and can be determined by querying the GL_EXTENSIONS string returned by *glGetString()* for the names of the extensions that implement the instruments.

Using the Extension

This section discusses using the extension in the following subsections:

"Specifying the Buffer"

"Enabling, Starting, and Stopping Instruments"

"Measurement Format"

"Retrieving Information"

Specifying the Buffer

To specify a buffer in which to collect instrument measurements, call *glInstrumentsBufferSGIX()* with *size* set to the size of the buffer as a count of GLints. The function has the following prototype:

```
void glInstrumentsBufferSGIX( GLsizei size, GLint *buffer )
```

The buffer will be prepared in a way that allows it to be written asynchronously by the graphics pipeline.

If the same buffer was specified on a previous call, the buffer is reset; that is, measurements taken after the call to *glInstrumentsBufferSGIX()* are written to the start of the buffer.

If *buffer* is zero, then any resources allocated by a previous call to prepare the buffer for writing will be freed. If *buffer* is non−zero, but is different from a previous call, the old buffer is replaced by the new buffer and any allocated resources involved in preparing the old buffer for writing are freed.

The buffer address can be queried with *glGetPointerv()* using the argument GL_INSTRUMENT_BUFFER_POINTER_SGIX (note that *glGetPointerv()* is an OpenGL 1.1 function).

Enabling, Starting, and Stopping Instruments

To enable an instrument, call *glEnable()* with an argument that specifies the instrument. The argument to use for a particular instrument is determined by the OpenGL extension that supports that instrument. (See "Instruments Example Pseudo Code".)

To start the currently enabled instrument(s), call *glStartInstrumentsSGIX()*. To take a measurement, call *glReadInstrumentsSGIX()*. To stop the currently−enabled instruments and take a final measurement call *glStopInstrumentsSGIX()*. The three functions have the following prototypes:

```
void glStartInstrumentsSGIX( void )
void glReadInstrumentsSGIX( GLint marker )
void glStopInstrumentsSGIX( GLint marker )
```

The *marker* parameter is passed through the pipe and written to the buffer to ease the task of interpreting it.

If no instruments are enabled executing *glStartInstrumentsSGIX()*, *glStopInstrumentsSGIX()*, or *glReadInstruments()* will not write measurements to the buffer.

Measurement Format

The format of any instrument measurement in the buffer obeys certain conventions:

The first word of the measurement is the *glEnable()* enum for the instrument itself.

The second word of the measurement is the size in GLints of the entire measurement. This allows any parser to step over measurements with which it is unfamiliar. Currently there are no implementation−independent instruments to describe.

Implementation−dependent instruments are described in the Machine Dependencies section of the reference page for glInstrumentsSGIX. Currently, only InfiniteReality systems support any extensions.

In a single measurement, if multiple instruments are enabled, the data for those instruments can appear in the buffer in any order.

Retrieving Information

To query the number of measurements taken since the buffer was reset, call *glGet()* using GL_INSTRUMENT_MEASUREMENTS_SGIX.

To determine whether a measurement has been written to the buffer, call *glPollInstrumentsSGIX()*, which has the following prototype:

```
GLint glPollInstrumentsSGIX( GLint *markerp )
```

If a new measurement has appeared in the buffer since the last call to *glPollInstrumentsSGIX()*, 1 is returned, and the value of marker associated with the measurement by *glStopInstrumentsSGIX()* or *glReadInstrumentsSGIX()* is written into the variable referenced by *marker_p*. The measurements appear in the buffer in the order in which they were requested. If the buffer overflows, *glPollInstrumentsSGIX()* may return −1 as soon as the overflow is detected, even if the measurement being polled did not cause the overflow. (An implementation may also choose to delay reporting the overflow until the measurement that caused the overflow is the one being polled.) If no new measurement has been written to the buffer, and overflow has not occurred, *glPollInstrumentsSGIX()* returns 0.

Note that while in practice an implementation of the extension is likely to return markers in order, this functionality is not explicitly required by the specification for the extension.

To get a count of the number of new valid GLints written to the buffer, call *glGetInstrumentsSGIX()*, which has the following prototype:

```
GLint glGetInstrumentsSGIX( void )
```

The value returned is the number of GLints that have been written to the buffer since the last call to *glGetInstrumentsSGIX()* or *glInstrumentsBufferSGIX()*. If the buffer has overflowed since the last call to *glGetInstrumentsSGIX()*, −1 is returned for the count. Note that*glGetInstrumentsSGIX()* can be used independently of *glPollInstrumentsSGIX()*.

Instruments Example Pseudo Code

Example 11−1 Instruments Example Pseudo Code

```
#ifdef GL_SGIX_instruments
          #define MARKER1 1001
          #define MARKER2 1002
          {
             static GLint buffer[64];
             GLvoid *bufp;
             int id, count0, count1, r;
             /* define the buffer to hold the measurements */
             glInstrumentsBufferSGIX(sizeof(buffer)/sizeof(GLint), b
uffer);

             /* enable the instruments from which to take measuremen
ts */
             glEnable(<an enum for a supported instrument, such as
                      GL_IR_INSTRUMENT1_SGIX>);

             glStartInstrumentsSGIX();
             /* insert GL commands here */
             glReadInstrumentsSGIX(MARKER1);
             /* insert GL commands here */
             glStopInstrumentsSGIX(MARKER2);

             /* query the number of measurements since the buffer wa
s specified*/
             glGetIntegerv(GL_INSTRUMENT_MEASUREMENTS_SGIX,&r);
                 /* now r should equal 2 */

             /* query the pointer to the instrument buffer */
             glGetPointervEXT(GL_INSTRUMENT_BUFFER_SGIX,&bufp);
                 /* now bufp should be equal to buffer */

             /*
              * we can call glGetInstrumentsSGIX before or after the
 calls to
              * glPollInstrumentsSGIX but to be sure of exactly what
              * measurements are in the buffer, we can use PollInstr
umentsSGIX.
              */
             count0 = glGetInstrumentsSGIX();
             /* Since 0, 1, or 2 measurements might have been return
ed to
              * the buffer at this point, count0 will be 0, 1, or 2
times
              * the size in GLints of the records returned from the
              * currently−enabled instruments.
              * If the buffer overflowed, count0 will be −1.
              */

             while (!(r = glPollInstrumentsSGIX(&id))) ;
             /* if r is −1, we have overflowed.  If it is 1, id will
              * have the value of the marker passed in with the firs
t
              * measurement request (should be MARKER1).  While it i
```

```
s 0,
              * no measurement has been returned (yet).
              */

             while (!(r = glPollInstrumentsSGIX(&id))) ;
             /* see the note on the first poll; id now should equal
MARKER2 */


             count1 = glGetInstrumentsSGIX();
             /* the sum of count0 and count1 should be 2 times the s
ize in GLints
              * of the records returned for all instruments that we
have enabled.
              */
          }
          #endif
```

New Functions

glInstrumentsBufferSGIX, glStartInstrumentsSGIX, glStopInstrumentsSGIX, glReadInstrumentsSGIX, glPollInstrumentsSGIX, glGetInstrumentsSGIX

Chapter 12
OpenGL Tools

This chapter explains how to work with these OpenGL tools:

"ogldebug—the OpenGL Debugger" lets you use a graphical user interface to trace and examine OpenGL calls. See page 269

"glc—the OpenGL Character Renderer" lets you render characters in OpenGL programs. See page 283.

"gls—The OpenGL Stream Utility" is a facility for encoding and decoding streams of 8−bit bytes that represent sequences of OpenGL commands. See page 283.

"glxInfo—The glx Information Utility" provides information on GLX extensions and OpenGL capable visuals, and the OpenGL renderer of an X server. See page 285.

ogldebug—the OpenGL Debugger

This section explains how to debug graphics applications with the OpenGL debugging tool ogldebug. The following topics are discussed:

"ogldebug Overview"

"Getting Started With ogldebug"

"Creating a Trace File to Discover OpenGL Problems"

"Interacting With ogldebug"

"Using a Configuration File"

"Using Menus to Interact With ogldebug"

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:

Set a breakpoint for all occurrences of a given OpenGL call.

Step through (or skip) OpenGL calls.

Locate OpenGL errors.

For a selected OpenGL context, display information about OpenGL state, current display lists, and the window that belongs to the application you are debugging.

Create a history ("trace") file of all OpenGL calls made. The history file is a gls file that contains comments and performance hints. You can convert it to legal C code using ogldebug command line options.

Note: If you are debugging a multiwindow or multicontext application, ogldebug starts a new session (a new window appears) each time the application starts a new process. In each new window, the process ID is displayed in the title bar.

The OpenGL debugger is not a general−purpose debugger. Use *dbx* and related tools such as cvd (CASEVision/Workshop Debugger) to find problems in the nonOpenGL portions of a program.

How ogldebug Operates

The OpenGL debugger works like this:

You invoke ogldebug for an application using the appropriate command line options (see "ogldebug Command−Line Options").

A special library (*libogldebug.so*) intercepts all OpenGL calls using the OpenGL streams mechanism. It interprets calls to OpenGL only and filters GLU, GLC, and GLX calls. GLU calls are parsed down to their OpenGL calls; the actual GLU calls are lost.

You can run, halt, step, and trace each process in the application separately using the ogldebug graphical interface.

After ogldebug−related processing, the actual OpenGL calls are made as they would have been if ogldebug had not been present.

Getting Started With ogldebug

This section discusses how to set up and start ogldebug and lists available command line options.

Setting Up ogldebug

Before you can use ogldebug, you must install the *gl_dev.sw.ogldebug* (or *gl_dev.sw64.debug*) subsystem. You can use the Software Manager from the Toolchest or execute *swmgr* from the command line. Consider also installing *gl_dev.man.ogldebug* to have access to the reference page.

ogldebug Command−Line Options

The ogldebug version that is shipped with IRIX 6.5 has a number of command−line options. (The options are also listen in the ogldebug reference page).

Table 12−1 Command−Line Options for ogldebug

Option	Description
-display *display*	Set the display for the ogldebug user interface. If -display is not specified, ogldebug will use $DISPLAY.
-appdisplay *display*	Set the display for the application.
-glsplay *gls_trace_file*	Play back a gls trace file recorded by ogldebug. Note that a gls trace file is not standard C.
-gls2c *gls_trace_file*	Convert a gls trace file to a C code snippet. Output is to stdout.
-gls2x *gls_trace_file*	Convert a gls trace file to an X Window System program that can be compiled. Output is to stdout.
-gls2glut *gls_trace_file*	Convert a gls trace file to a GLUT program that can be compiled. Output is to stdout.

Starting ogldebug

To debug your OpenGL program, type the appropriate command line for your executable format:

o32	% **`ogldebug`***optionso32program_name program_options*
n32	% **`ogldebug32`***optionsn32program_name program_options*
64	% **`ogldebug64`***options64program_name program_options*

where

options are any of the options listed under "ogldebug Command−Line Options."

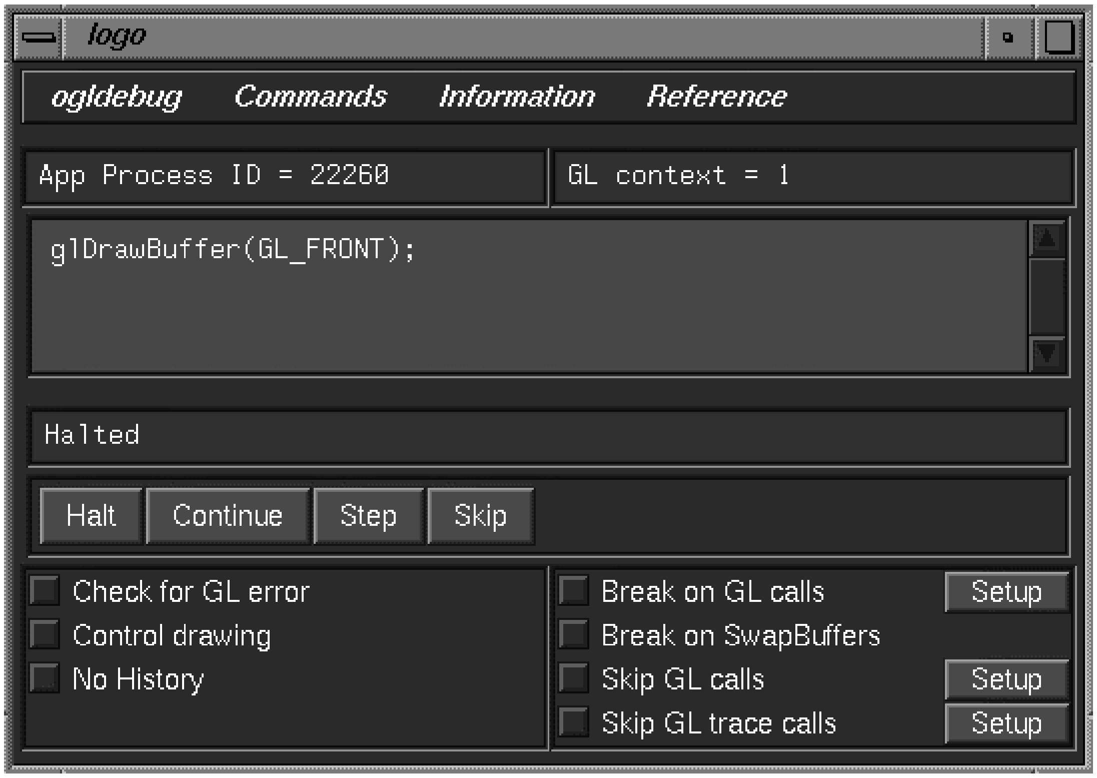
program_name is the name of your (executable) application.

program_options are application−specific options, if any.

Note: It is not necessary to compile the application with any special options. The debugger works with any program compiled with **-lGL**.

ogldebug becomes active when the application makes its first OpenGL call. Each ogldebug main window represents a different application process. If the application uses fork, sproc, or pthread, multiple ogldebug windows may appear.

The debugger launches your application and halts execution just before the application's first OpenGL call. The main window (see Figure 12−1) lets you interact with your application's current process and displays information about the process.



App Process ID = 22260
GL context = 1

```
glDrawBuffer(GL_FRONT);
```

Halted

Check for GL error
Control drawing
No History
Break on GL calls
Break on SwapBuffers
Skip GL calls
Skip GL trace calls

Figure 12−1ogldebug Main Window

The three display areas below the menu bar are:

Context information. Displays the current process for that window (multiple processes have multiple windows) and the current OpenGL context.

OpenGL call display. Below the status display area is the OpenGL call display area. This area shows the next OpenGL command to be executed.

Status display. Immediately above the row of buttons is a one−line status display field, where ogldebug posts confirmation of commands and other status indicators.

Below the row of buttons are checkboxes, discussed in "Using Checkboxes".

Interacting With ogldebug

This section provides more detailed information on working with ogldebug. It explains.

"Commands for Basic Interaction"

"Using Checkboxes"

Additional information is available in the sections "Creating a Trace File to Discover OpenGL Problems" and "Using Menus to Interact With ogldebug".

Commands for Basic Interaction

You can perform all basic interaction using the row of buttons just above the check boxes. 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 12−2 Command Buttons and Shortcuts

Command	Result
Halt **Ctrl+H**	Temporarily stops the application at the next OpenGL call. All state and program information is retained so you can continue execution if you wish.
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. The display will only be updated when the application stops again.
Step **Ctrl +T**	Continues executing up to the next OpenGL call, then stops before executing that call.
Skip **Ctrl +K**	Skips over the current OpenGL call. Useful if you think the current call contains an error or is likely to cause one. The program executes until it reaches the next OpenGL call, then stops.

Using Checkboxes

The checkboxes at the bottom of the ogldebug window allow finer control over how information is collected. Checkboxes let you determine when a break occurs and which API calls you want to skip.

Table 12−3explains what happens for each of these boxes if it is checked.

Table 12−3 ogldebug Check Boxes

Check box	Description
Check for GL error	Calls *glGetError()* after every OpenGL call to check for errors. Note that *glGetError()* cannot be called between *glBegin()* and *glEnd()* pairs. *glGetError()* is called until all errors are clear.
Control drawing	Allows the user to inspect drawing in progress (forces front buffer rendering). Also allows the user to control drawing speed.
No history	Does not record history of the OpenGL call. As a result, the program

	runs faster but you cannot look at history information.
Break on GL calls	Halts on selected Open GL calls. Use the adjacent Setup button to select which calls to skip (see Figure 12−2). In the "Break on GL calls" Setup box, *glFlush()* is selected by default but is not active unless the "Break on GL calls" checkbox is selected.
Break on SwapBuffers	Halts oncalls that swap buffers. There is no window system independent call that swaps buffers; the debugger halts on the appropriate call for each platform (e.g. *glxSwapBuffers()* for X Window System applications).
Skip GL calls	Skips selected OpenGL calls. Use the adjacent *Setup* button to select which calls to skip.
Skip GL trace calls	Does not write selected OpenGL calls to the trace file. Use the adjacent *Setup* button to select which calls you don't want traced.

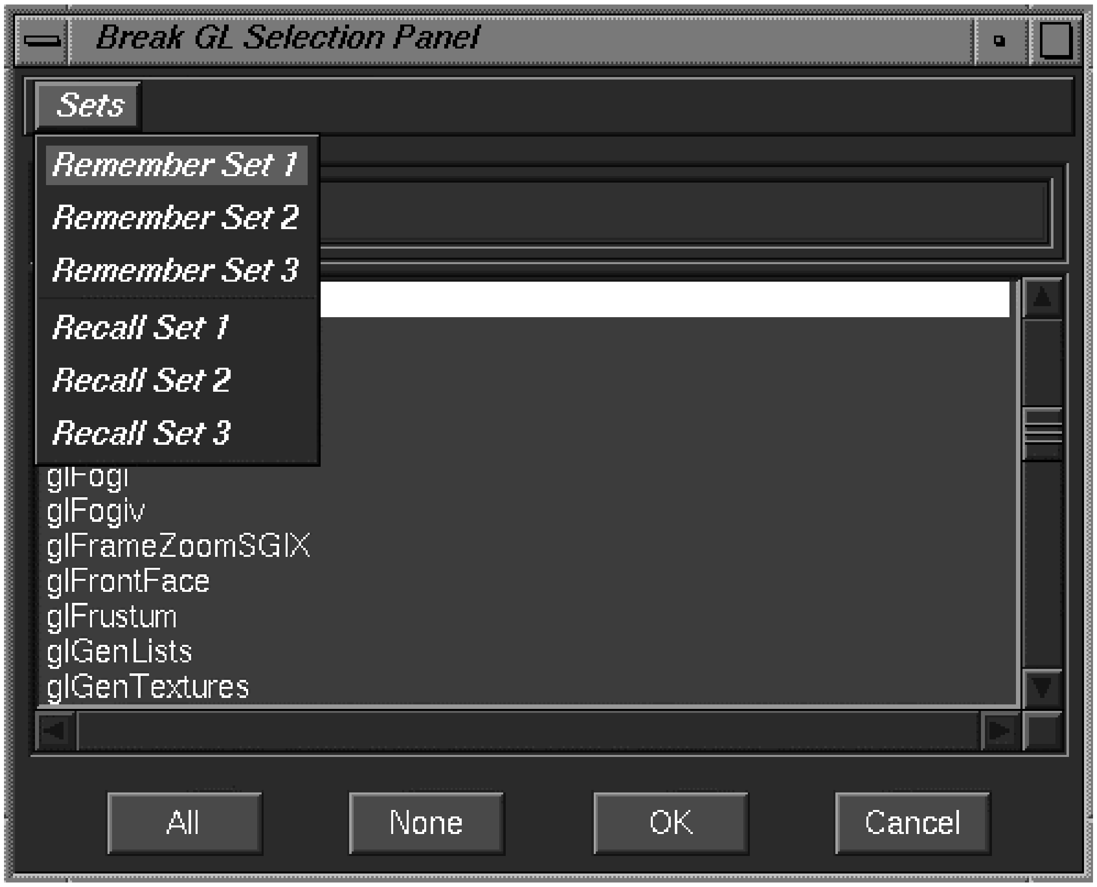


Figure 12−2 Setup Panel

Figure 12−2shows a setup panel. Inside any setup panels, you can use the standard Shift, Control, and Shift+Control keystrokes for multiple item selection and deselection.

To save and recall up to three custom selection/deselection areas, use the Sets menu in the setup panel for Break on OpenGL calls, Skip GL calls, and Skip GL trace calls.

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 is an example of how to collect one frame of OpenGL calls:

1. Launch ogldebug:

```
% ogldebug your_program_name
```

Be sure to use the appropriate options, see "ogldebug Command−Line Options".

2. Run until the application has passed the point of interest. You can do either of these substeps:
   - Click the Break on SwapBuffers checkbox
   - Click the Break (API calls) checkbox to select it, then click the Setup button next to it and choose *glFlush()* in the Break Selection panel.
3. From the Information menu, select Call History.

   ogldebug presents a panel that lets you select which OpenGL context you want to trace. Depending on the application, more than one context may be available.
4. Select the OpenGL context you want to trace.

   A Call History panel appears, showing a list of all OpenGL contexts in the application. Double−clicking the context will show an additional window with all calls from that context. You can examine the call history in the panel or save it as a gls trace file using the *Save* button at the bottom of the panel.

   A gls trace is meant to be pure OpenGL and to be window−system independent. Comments have, however, been added that indicate where GLX, GLU, and GLC calls were made. Any OpenGL calls made from within these higher−level calls are indented. Performance hints are also included in the trace file, as in the following example:

```
...

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
glMaterialfv(GL_FRONT, GL_AMBIENT, {0.1745, 0.01175, 0.01175, 2.5
89596E−29});
glsString("Info", "For best performance, set up material paramete
rs first, then enable lighting.");

...
```

5. At this point, you have several options:
   - Play back (re−execute) the gls trace file with the **−glsplay** option.
   - Convert the gls trace file to a C file by invoking ogldebug with the **−gls2c**, **−gls2x**, or **−gls2glut** option. Any comments or performance hints are removed during the conversion.

For larger applications, such as Performer, consider using the No History feature. If you need to run the application to a particular point and do not care about the call history until that point, turn on "No history" to speed things up.

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.

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

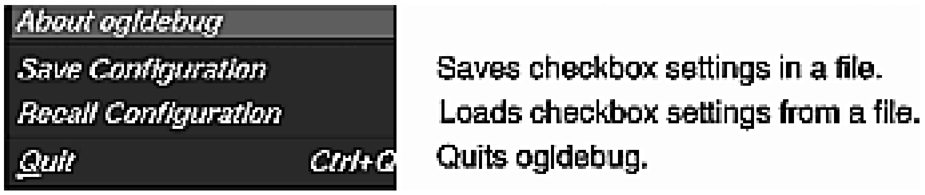
Using the Commands Menu to Interact With Your Program

Using the Information Menu to Access Information

Using the References Menu for Background Information

Using the File Menu to Interact With ogldebug

The File menu (shown in Figure 12−3) gives version information, lets you save and reload a configuration file, and quits ogldebug.



Saves checkbox settings in a file.
Loads checkbox settings from a file.
Quits ogldebug.

Figure 12−3ogldebug File Menu

Using the Commands Menu to Interact With Your Program

The Commands menu gives access to some of the information collected by ogldebug.The commands are discussed in "Interacting With ogldebug".

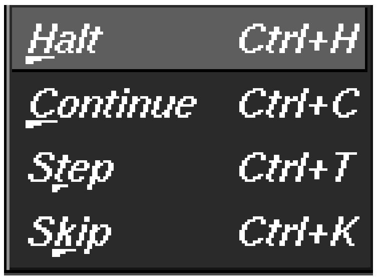
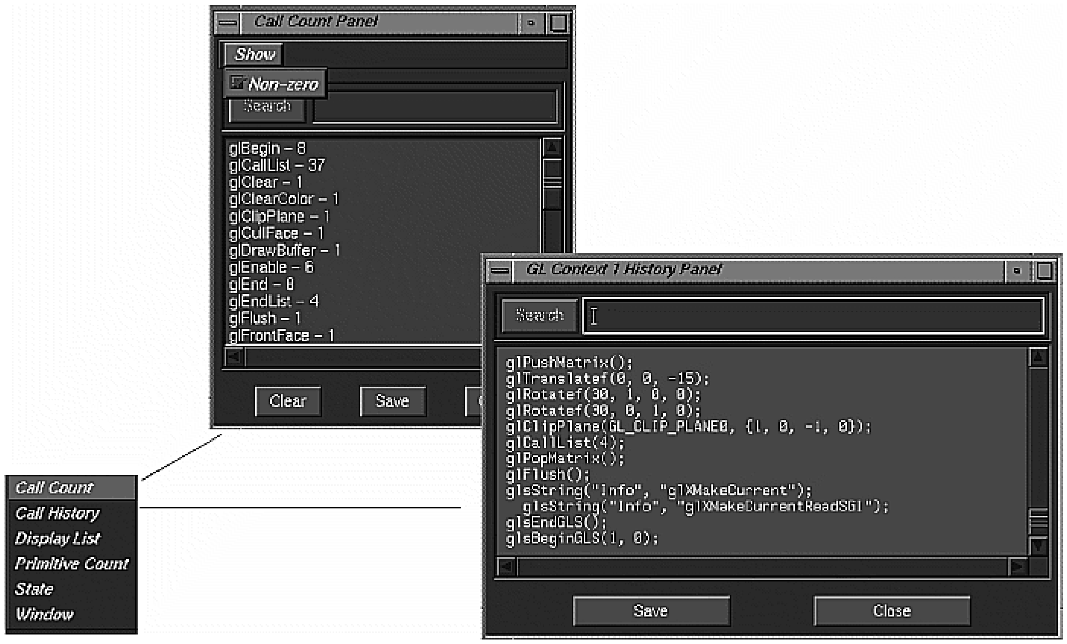


Figure 12−4ogldebug Command menu

Using the Information Menu to Access Information

The following two illustrations show the windows in which ogldebug displays information. A table that explains the functionality follows each illustration.



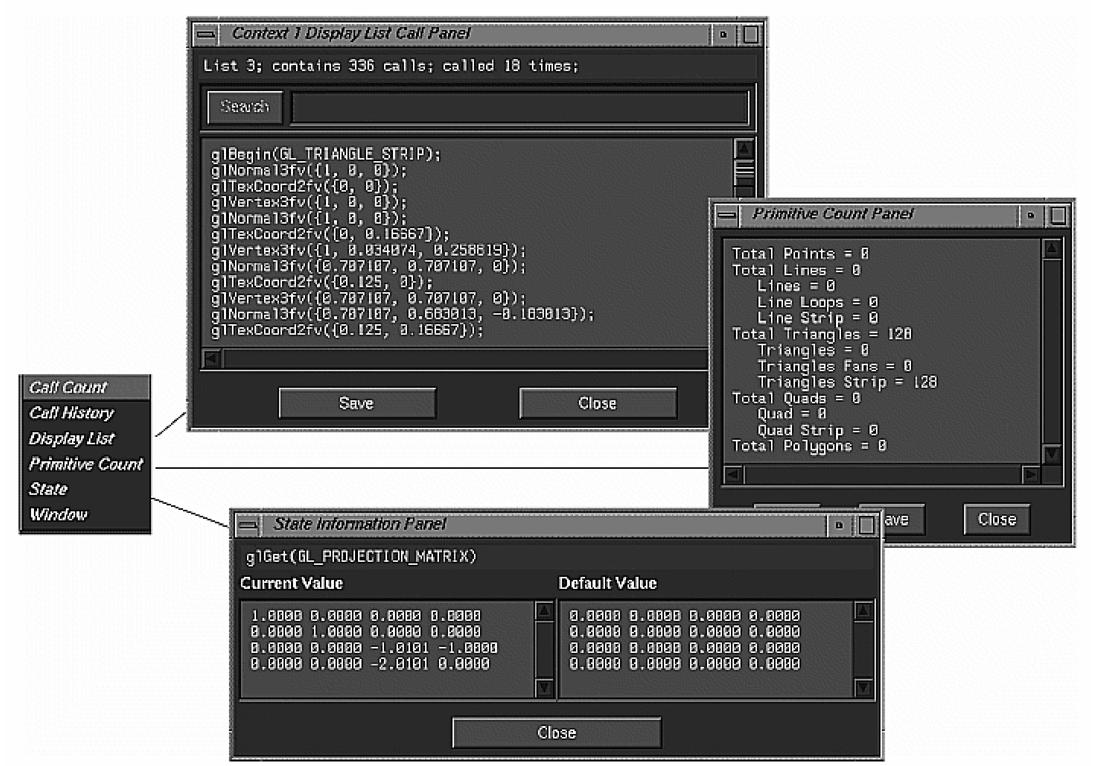
```
glBegin – 8
glCallList – 37
glClear – 1
glClearColor – 1
glClipPlane – 1
glCullFace – 1
glDrawBuffer – 1
glEnable – 6
glEnd – 8
glEndList – 4
glFlush – 1
glFrontFace – 1
```

```
glPushMatrix();
glTranslatef(0, 0, -15);
glRotatef(30, 1, 0, 0);
glRotatef(30, 0, 1, 0);
glClipPlane(GL_CLIP_PLANE0, {1, 0, -1, 0});
glCallList(4);
glPopMatrix();
glFlush();
glsString("Info", "glXMakeCurrent");
  glsString("Info", "glXMakeCurrentReadSGI");
glsEndGLS();
glsBeginGLS(1, 0);
```

Figure 12−5 Information Menu Commands (First Screen)

Here's a brief description of the Call Count and Call History menu commands:

Call Count	Brings up a window with counts for OpenGL, GLU, GLX, and GLC calls. You can s a count for all OpenGL functions or only for functions that were called at least once (nonzero calls).
Call History	Brings up a window with a history of OpenGL calls (as a gls trace).



```
List 3; contains 336 calls; called 18 times;
```

```
glBegin(GL_TRIANGLE_STRIP);
glNormal3fv({1, 0, 0});
glTexCoord2fv({0, 0});
glVertex3fv({1, 0, 0});
glNormal3fv({1, 0, 0});
glTexCoord2fv({0, 0.16667});
glVertex3fv({1, 0.034074, 0.258819});
glNormal3fv({0.707107, 0.707107, 0});
glTexCoord2fv({0.125, 0});
glVertex3fv({0.707107, 0.707107, 0});
glNormal3fv({0.707107, 0.683013, -0.183013});
glTexCoord2fv({0.125, 0.16667});
```

```
Total Points = 0
Total Lines = 0
   Lines = 0
   Line Loops = 0
   Line Strip = 0
Total Triangles = 128
   Triangles = 0
   Triangles Fans = 0
   Triangles Strip = 128
Total Quads = 0
   Quad = 0
   Quad Strip = 0
Total Polygons = 0
```

```
glGet(GL_PROJECTION_MATRIX)
```

```
Current Value                      Default Value
1.0000 0.0000 0.0000 0.0000        0.0000 0.0000 0.0000 0.0000
0.0000 1.0000 0.0000 0.0000        0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 -1.0101 -1.0000      0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 -2.0101 0.0000       0.0000 0.0000 0.0000 0.0000
```

Figure 12−6 Information Menu Commands (Second Screen)

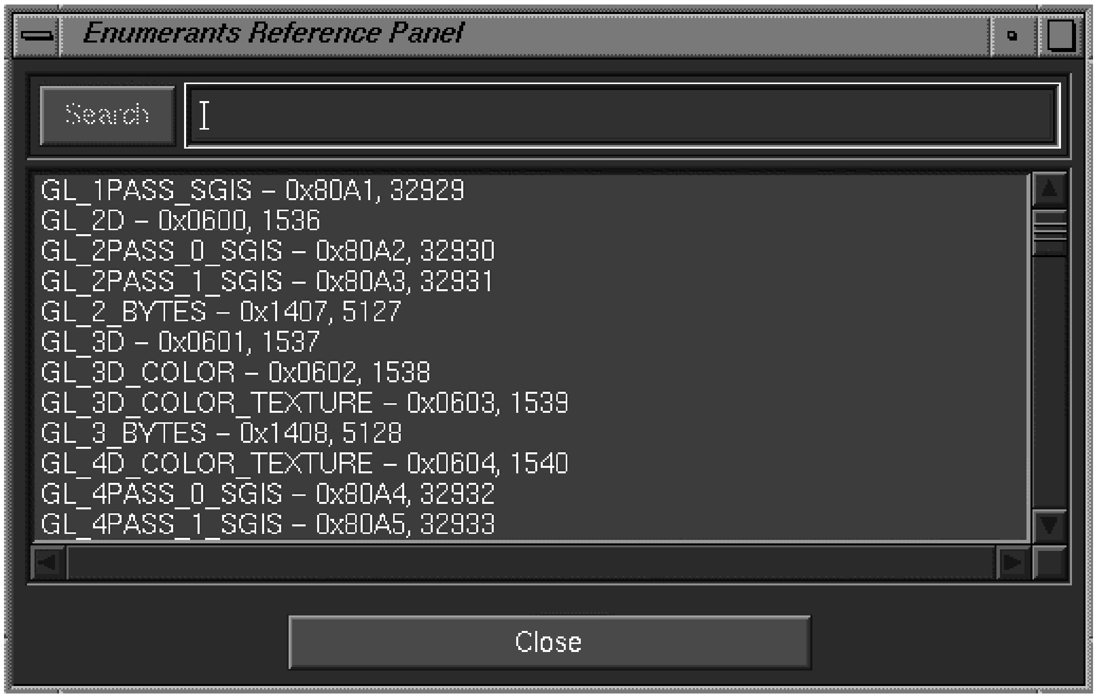
Here is a brief description of the menu commands:

Display List	First prompts for a context, then brings up a window with information about the

	application’s display lists, if any, for that context. You can show all or only non−em display lists.
Primitive Count	Provides the number of all primitives sent by the application so far (for example, qua polygons, and so on). Whether they are clipped or not is not reported.
State	Brings up a window that displays information on OpenGL state variables. You can s all or only nondefault state. Note that you cannot query state between *glBegin()* and *glEnd()* pairs.
Window (not shown)	Brings up window information for the application you are running from ogldebug.

Using the References Menu for Background Information

The References menu provides access to the Enumerants menu command only. 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 (See Figure 12−7).



```
GL_1PASS_SGIS − 0x80A1, 32929
GL_2D − 0x0600, 1536
GL_2PASS_0_SGIS − 0x80A2, 32930
GL_2PASS_1_SGIS − 0x80A3, 32931
GL_2_BYTES − 0x1407, 5127
GL_3D − 0x0601, 1537
GL_3D_COLOR − 0x0602, 1538
GL_3D_COLOR_TEXTURE − 0x0603, 1539
GL_3_BYTES − 0x1408, 5128
GL_4D_COLOR_TEXTURE − 0x0604, 1540
GL_4PASS_0_SGIS − 0x80A4, 32932
GL_4PASS_1_SGIS − 0x80A5, 32933
```

Figure 12−7 Enumerants Window

glc—the OpenGL Character Renderer

The OpenGL Character Renderer (GLC) is a platform−independent character renderer that offers the following benefits:

- Convenient to use for simple applications
- Can scale and rotate text and draw text using lines, filled triangles, or bitmaps)
- Supports for international characters

For a basic discussion of glc and a list of notes and known bugs for the current implementation, see the glcintro reference page.

The most authoritative documentation on GLC is the GLC specification document, which is usually included in each OpenGL release in PostScript form. If you install the software product *gl_dev.sw.samples*, the GLC specification is installed as

```
/usr/share/src/OpenGL/teach/glc/glcspec.ps
```

gls—The OpenGL Stream Utility

The OpenGL Stream Codec (GLS) is a facility for encoding and decoding streams of 8−bit bytes that represent sequences of OpenGL commands. This section starts with an overview of gls, then discusses "glscat Utility", which allows you to concatenate gls streams.

OpenGL Stream Utility Overview

GLS can be used for a variety of purposes, for example:

Scalable OpenGL pictures—GLS facilitates resolution−independent storage, interchange, viewing, and printing.

Persistent storage of OpenGL commands, display lists, images, and textures.

Communication—Command transfer between application processes via byte−stream connections.

Client−side display lists—Can contain client data or callbacks.

Tracing—Useful for debugging, profiling, and benchmarking.

Some of these applications require the definition and implementation of higher−level APIs that are more convenient to use than the GLS API. The GLS API provides only the basic encoding and decoding services that allow higher−level services to be built on top of it efficiently.

The GLS specification has two components:

A set of three byte−stream encodings for OpenGL and GLS commands: human−readable text, big−endian binary, and little−endian binary. The three encodings are semantically identical; they differ only in syntax. It is therefore possible to convert GLS byte streams freely among the three encodings without loss of information.

An API that provides commands for encoding and decoding GLS byte streams. This API is not formally an extension of the OpenGL API. Like the GLU API, the GLS API is designed to be implemented in an optional, standalone client−side subroutine library that is separate from the subroutine library that implements the OpenGL API.

The GLS encodings and API are platform independent and window system independent. In particular, the GLS encodings are not tied to the X Window System protocol encoding used by the GLX extension. GLS is designed to work equally well in UNIX, Windows, and other environments.

For information, see the glsintro reference page.

glscat Utility

The *glscat* utility (*/usr/sbin/glscat*) allows you to concatenate GLS streams. Enter **`glscat −h`** at the command line for a list of command−line parameters and options for*glscat*.

In its simplest usage, *glscat* copies a GLS stream from standard input to standard output:

```
glscat < stream1.gls > stream2.gls
```

As an alternative to standard input, one or more named input files can be provided on the command line. If multiple input streams are provided, GLS will concatenate them:

```
glscat stream1.gls stream2.gls > stream3.gls
```

Use the **−o** *outfile* option to specify a named output file as an alternative to standard output:

glscat -o *stream2.gls* < *stream1.gls*

In all cases, the input stream is decoded and re−encoded, and errors are flagged. By default, the type of the output stream (GLS_TEXT, GLS_BINARY_MSB_FIRST, or GLS_BINARY_LSB_FIRST) is the same as the type of the input stream.

The most useful option to glscat is the **−t** *type*, which lets you control the type of the output stream. The *type* parameter is a single−letter code, one of the following:

t	Text
b	Native binary
s	Swapped binary
l	lsb−first binary
m	msb−first binary

For example, the following command converts a GLS stream of any type to text format:

```
glscat −t t < stream1.gls > stream2.gls
```

glxInfo—The glx Information Utility

glxinfo lists information about the GLX extension, OpenGL capable visuals, and the OpenGL renderer of an X server. The GLX and render information includes the version and extension attributes. The visual information lists the GLX visual attributes for each OpenGL capable visual (for example whether the visual is double buffered, the component sizes, and so on). For more information, try out the command or see the glxinfo reference page.

Chapter 13
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:

"Debugging and Tuning Your Program"

"General Tips for Debugging Graphics Programs"

"About Pipeline Tuning"

"Tuning Animation"

"Taking Timing Measurements"

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 14, "Tuning the Pipeline"

Chapter 15, "Tuning Graphics Applications: Examples"

Chapter 16, "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".

Debugging and Tuning Your Program

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, Silicon Graphics systems let you make tradeoffs between image quality and performance for your application.

This section sets the foundation for good performance by discussing:

"General Tips for Debugging Graphics Programs"

"Specific Problems and Troubleshooting"

General Tips for Debugging Graphics Programs

This section gives advice on important aspects of OpenGL debugging. Most points apply primarily to graphics programs and may not be obvious to developers who are accustomed to debugging text−based programs.

Here are some general debugging tips for 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. See "ogldebug—the OpenGL Debugger" for more information on ogldebug.

Use an iterative coding process: add some graphics−related code, build and test to ensure expected results, and repeat as necessary.

Debug the parts of your program in order of complexity: First make sure your geometry is drawing correctly, then add lighting, texturing, and backface culling.

Start debugging in single−buffer mode, then move on to a double−buffered program.

Here are some areas that frequently result in errors:

Be careful with OpenGL enumerated constants that have similar names. 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.

Check for matching *glPushMatrix()* and *glPopMatrix()* calls.

Check matrix mode state information. Generally, an application should stay in GL_MODELVIEW mode. Odd visual effects can occur if the matrix mode is not right.

Specific Problems and Troubleshooting

This section discusses 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".

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 are bound to the right window, try clearing the image buffers with *glClear()*. If you cannot clear, you may be bound to the wrong window (or no window at all).

To make sure you are not rendering in the background color, use an unusual color (instead of black) to clear the window with *glClear()*.

To make sure you are 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 up the viewpoint up to see more of the space.

Check the section "Troubleshooting Transformations" in Chapter 3 of the *OpenGL Programming Guide, Second Edition*.

Make sure you are using the correct projection matrix.

Remember that *glOrtho()* and *glPerspective()* calls multiply onto the current projection matrix; they don't replace it.

If you have a blank window in a double−buffered program, check first that something is displayed when you run the program in single−buffered mode. If yes, make sure you are calling *glXSwapBuffers()*. If the program is using depth buffering and that the depth buffer is cleared as appropriate. See also "Depth Buffering Problems".

Check the aspect ratio of the viewing frustrum. Don't set up your program using code like the following:

```
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 have translated back to the origin before rotating (unless you intend to rotate about some other point). Rotations are always about the origin of the current coordinate system.

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. If you don't use resources to control visual selection, call *glDrawBuffer()* with a GL_FRONT argument before rendering. You can then observe the image as it is drawn. Note that this observation is possible only if the problem is severe. On a fast system you may not be able to observe the problem.

Lighting Problems

Turn off specular shading in the early debugging stages. It is harder to visualize where specular highlights should be than where diffuse highlights should be.

For local light sources, draw lines from the light source to the object you are trying to light to make sure the spatial and directional nature of the light is right.

Make sure you have both GL_LIGHTING enabled and the appropriate GL_LIGHT#'s enabled.

To see whether normals are being scaled and causing lighting problems, enable GL_NORMALIZE. 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 calling`glLight(GL_POSITION)` provides a headlight if called before *gluLookAt()* and so on.

X Window System Problems

OpenGL and the X Window System have different notions of the *y* direction. OpenGL has it in the lower left corner of the window; X has the origin (0, 0) in the upper 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.

Textures and display lists defined in one context are not visible to other contexts unless they explicitly share textures and display lists.

glXUseXFont() creates display lists for characters. The display lists are visible only in contexts that share objects with the context in which they were created.

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. If this value is not set correctly, the image looks sheared.

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 the system you are targeting supports destination alpha planes. Some low−end machines don't support them.

About 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. If different parts of the hardware are responsible for different parts of the pipeline, 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.

The goal of any program is a balanced pipeline; highest−quality rendering at optimum speed. Different programs stress different parts of the pipeline, so it is important to understand which elements in the graphics pipeline are a program's bottlenecks.

Three−Stage Model of the Graphics Pipeline

The graphics pipeline in all Silicon Graphics workstations consists of three conceptual stages (see Figure 13−1): Depending on the implementation, all parts may be done by the CPU or parts of the pipeline may be done by an accelerator card. The conceptual model is useful in either case: it helps you to understand where your application spends its time. These are the stages:

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 (may be hardware accelerated).
3. **The raster subsystem.** The per−pixel and per−fragment 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.

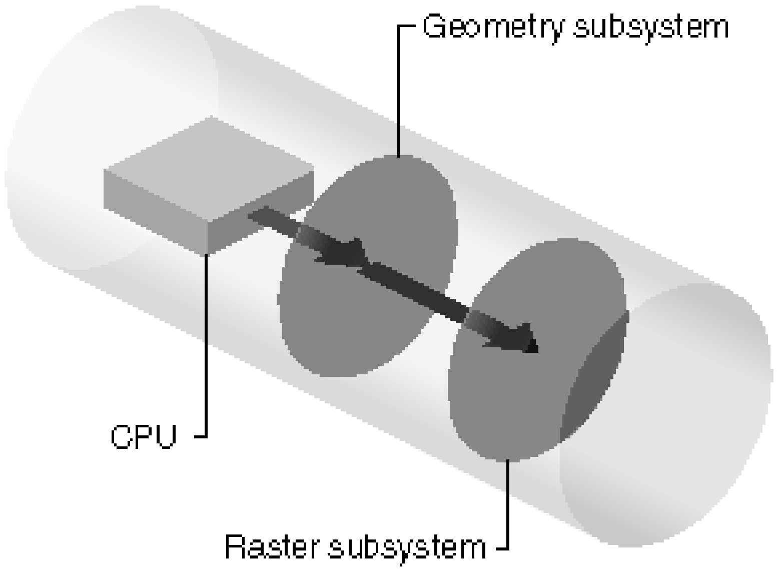


Figure 13−1 Three−Stage Model of the Graphics Pipeline

Note 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 geometry elements such that they remain the same size but look more detailed because they are composed of a larger number of polygons.

Note that in a *software implementation,* all the work is done on the host CPU. As a result, it doesn't make sense to increase the work in the geometry pipeline if rasterization is the bottleneck: you would increase the work for the CPU and decrease performance.

Isolating Bottlenecks in Your Application: 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 do not 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 noticeably, that stage is not the bottleneck. If there is a noticeable difference in performance, you have 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 see if your application is the 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".

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".

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 doesn'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 or by removing clear operations. If any of these experiments speeds up the program, it has a per-pixel bottleneck. For more information, see "Tuning the Raster Subsystem".

Usually, the following order of operations is most expedient:

1. First determine if your application is host (CPU) limited using *gr_osview* and checking whether the CPU usage is near 100%. The *gr_osview* program also includes statistics that indicate whether the performance bottleneck is in the graphics subsystem or in the host.
2. Then check whether the application is fill (per−pixel) limited by shrinking the window.
3. If the application is neither CPU limited nor fill limited, you have to prove that it is geometry limited.

Note that on some systems you can have a bottleneck just in the transport layer between the CPU and the geometry. To test whether that is the case, try sending less data, for example call *glColor3ub()* instead of g*lColor3f().*

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 15, "Tuning Graphics Applications: Examples."

Factors Influencing Performance

Pipeline tuning is discussed in detail in Chapter 14, "Tuning the Pipeline."Table 13−1provides an overview of factors that may limit rendering performance and the part of the pipeline they belong to.

Table 13−1 Factors Influencing Performance

Performance Parameter	Pipeline Stage
Amount of data per polygon	All stages

Time of application overhead	CPU subsystem (application)
Transform rate & mode setting for polygon	Geometry subsystem
Total number of polygons in a frame	Geometry and raster subsystem
Number of pixels filled	Raster subsystem
Fill rate for the given mode settings	Raster subsystem
Time of color and/or depth buffer clear	Raster subsystem

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 close to the best the hardware can achieve, start following the more general tuning tips provided in this manual. The next step is, however, 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 minimum activity is taking place on your system while you take timing measurements. Other graphics programs, background processes, and network activity can distort timing results because they use system resources. For example, do not have *osview*, *gr_osview*, or *Xclock* running while you are benchmarking. If possible, turn off network access as well.

Work with local files.

Unless your goal is to time a program that runs on a remote system, make sure that all input and output files, including the file used to log results, are local.

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 is 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 is faster, write a loop in the example program 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 while the animation is running. Once a single frame has been analyzed and tuned, look at frames that stress the graphics pipeline in different ways, analyzing and tuning each frame.

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.

To get accurate numbers, you must perform timing trials in single−buffer mode, with no calls to *glXSwapBuffers()*.

Because buffers can be swapped only 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.

When making timing measurements, use *glFinish()* to ensure that all pixels have been drawn before measuring the elapsed time.

Benchmark programs should exercise graphics in a way similar to the actual application. In contrast to the actual application, the benchmark program should perform only graphics operations. Consider using ogldebug to extract representative OpenGL command sequences from the program. See "ogldebug—the OpenGL Debugger" for more information.

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 Your Application: Overview".

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.

Timing analysis takes effort. In practice, it is 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 perform 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 13−2shows 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 15, "Tuning Graphics Applications: Examples."

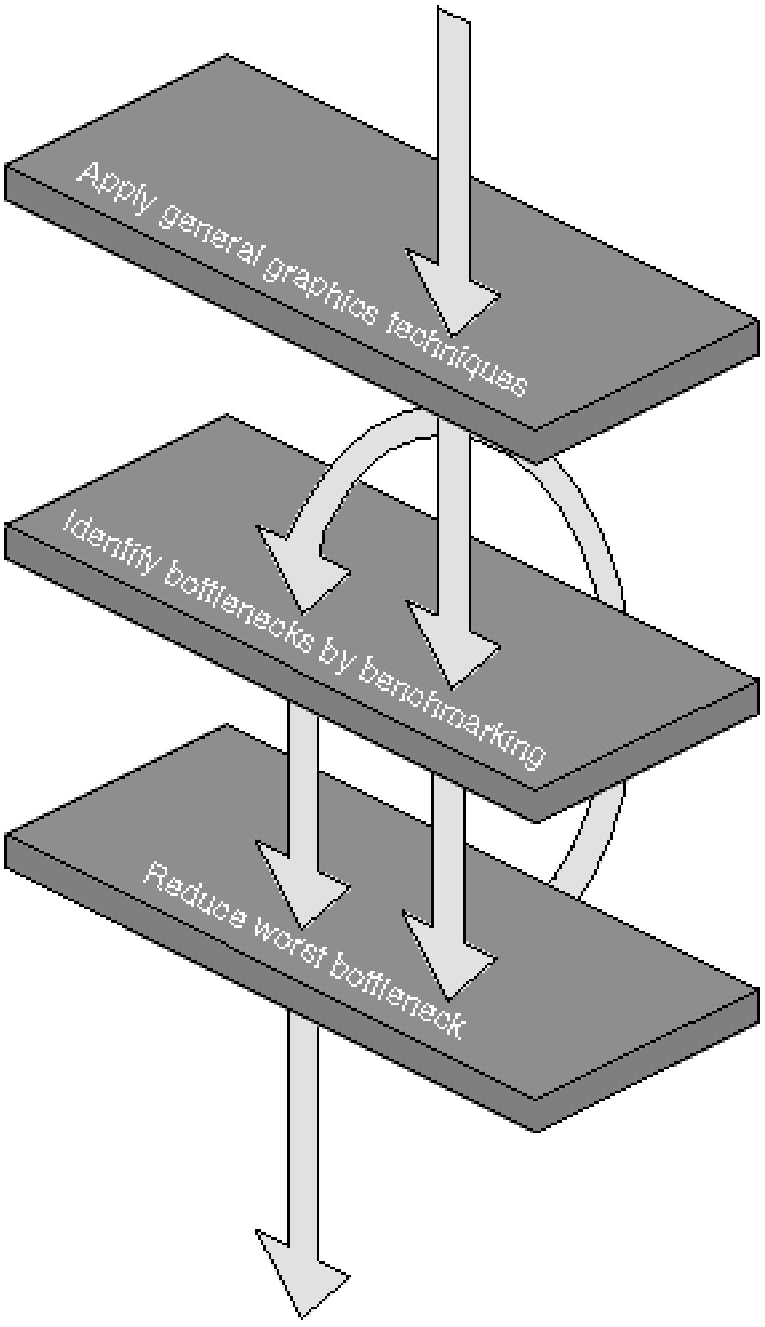

Apply general graphics techniques
Identify bottlenecks by benchmarking
Reduce worst bottleneck

Figure 13−2 Flowchart of the Tuning Process

Tuning Animation

Tuning animation requires attention to some factors not relevant in other types of applications. This section first explores how frame rates determine animation speed, then provides some advice for optimizing an animation’s performance.

Smooth animation requires double buffering. In double buffering, one framebuffer holds the current frame, which is scanned out to the monitor by the 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.

How Frame Rate Determines 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 each individual frame are shown in Table 13−1above.

When trying to improve animation speed, consider these points:

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.

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.

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, then benchmark how many frames you can draw. If you don’t, it is difficult to know if you are 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".

In addition, follow these guidelines to optimize frame rate performance:

Reduce drawing time to a lower multiple of the screen refresh time (16.7 msec on a 60 Hz monitor).

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.

Chapter 14

Tuning the Pipeline

This chapter discusses 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"
- "CPU Tuning: Immediate Mode Drawing"
- "CPU Tuning: Display Lists"
- "CPU Tuning: Advanced Techniques"
- "Tuning the Geometry Subsystem"
- "Tuning the Raster Subsystem"
- "Tuning the Imaging Pipeline"

CPU Tuning: Basics

The first stage of the rendering pipeline is traversal of the data and sending of 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 are 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.

  If you aren’t concerned about backward compatibility, compile for the n32 abi instead of compiling for o32. The default on IRIX 6.5 is n32.

- 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

When deciding whether you want to use display list or immediate mode drawing, consider the amount of work you do in constructing your databases and using them for purposes other than graphics. Here are three cases to consider:

> If you create models that never change, and are used only for drawing, then OpenGL display lists are the right representation.
>
> Display lists can be optimized in hardware−specific ways, loaded into dedicate display list storage in the graphics subsystem, downloaded to on−board dlist RAM, and so on. See"CPU Tuning: Display Lists" for more information on display lists.
>
> If you create models that are subject to infrequent change, but are rarely used for any purpose other than drawing, then vertex arrays are the right representation.
>
> Vertex Arrays are relatively compact and have modest impact on cache. Software renderers can process the vertices in batches; hardware renderers can trickle triangles out a few at a time to maximize parallelism. As long as the vertex arrays can be retained from frame to frame, so you do not incur a lot of latency by building them afresh each frame, they are the best solution for this case. See "Using Vertex Arrays" for more information.
>
> If you create very dynamic models, or if you use the data for heavy computations unrelated to graphics, then the *glVertex()*−style interface (immediate mode drawing) is the best choice.
>
> Immediate mode drawing allows you to maximize parallelism for hardware renderers and to optimize your database for the other computations you need to perform, and it reduces cache thrashing. Overall, this will result in higher performance than forcing the application to use a graphics−oriented data structure like a vertex array. Use immediate−mode drawing for large databases (which might have to be paged into main memory) and dynamic databases, for example for morphing operations where the number of vertices is subject to change, or for progressive refinement. See "CPU Tuning: Immediate Mode Drawing" for tuning information.

If you are still not sure whether to choose display lists or immediate mode drawing, consider the following advantages and disadvantages of display lists.

Display lists have the following advantages:

> You don’t have to optimize traversal of the data yourself; display list traversal is well−tuned and more efficient 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 Indigo2 IMPACT systems, see "Using Display Lists

Effectively".

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 application data plus additional overhead for each command. If the application has no other need for the data then drawing, it can release the storage for its copy of the data and the penalty is negligible.

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.

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.

For example, suppose you want to apply a transformation to some geometric objects and then draw the result. If the geometric objects are to be transformed in the same way each time, it is better to store the matrix in the display list. 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 t traversed again.This section explains how to optimize display lists.

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—watch out for scaling of normals.

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 results in a lookup operation to find the designated display list. A flat display list requires less memory and yields simpler and faster traversal. It also improves cache coherency.

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 16, "System−Specific Tuning,"for potential display list optimizations on the system you are using.

CPU Tuning: Immediate Mode Drawing

Immediate mode drawing means that OpenGL commands are executed when they are 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 of its exact location; 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.

Do not interlace data used to render frames and infrequently used data in memory. Instead, include a pointer to the infrequently used data and store the data itself elsewhere.

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

The appropriate amount of flattening depends on the system on which your application will run.

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.

During rendering, an application ideally 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:

Good	`glVertex3fv(object−>data−>vert);`
OK	`glVertex3fv(dataptr−>vert);`
Bad	`glVertex3fv(dataptr);`

The following code fragment is an example of efficient code to draw a single smooth−shaded, lit polygon. 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+3);
glNormal3fv(ptr+6);
glVertex3fv(ptr+9);
glNormal3fv(ptr+12);
glVertex3fv(ptr+15);
glNormal3fv(ptr+18);
glVertex3fv(ptr+21);
glEnd();
ptr += 24;
```

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 and R10000, 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+3);
glVertex3fv(dataptr+5);
```

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.

Subroutine prototyping. Prototype subroutines in ANSI C style to avoid runtime typecasting of parameters:

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

Multiple primitives. Send multiple primitives 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. These checks waste time and also bring all of the object data through the cache, reducing the performance advantage of 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 strip length may vary depending on the hardware the program runs on. For more information, see Chapter 16, "System−Specific Tuning."

The specialized strips are 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.

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.

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.

Memory Organization

On most systems, 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.

Minimizing 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. Starting with IRIX 6.5, each page consists of 16 KB. In earlier versions of IRIX, each page consists of 4 KB (16 KB in high−end systems). 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 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.

Memory access is much faster if the data is already loaded into the first−level cache. When your program accesses data that is not in one of the caches, a cache miss results. This causes a cache line of several bytes, including the data you just accessed, to be read from memory and stored in the cache. The size of this transaction varies from machine to machine. Caches are broken down into lines, typically 32−128 bytes. When a cache miss occurs, the corresponding line is loaded from the next level down in the hierarchy.

Because cache misses are costly, try to minimize them by following these steps:

Keep frequently accessed data together. Store and access frequently used data in flat, sequential files and structures and avoid pointer indirection. This way, the most frequently accessed data remains in the first−level cache wherever possible.

Access data sequentially. If you are accessing words sequentially, each cache miss brings in 32 or more words of needed data; if you are accessing every 32nd word, each cache miss brings in one needed word and 31 unneeded words, degrading performance by up to a factor of 32.

Avoid simultaneously traversing several large independent 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 interleaved 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. If you are using vertex arrays, try using interleaved arrays.

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" 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 execution profiling results collected with PC sampling with those of basic block counting, performing each test with and without calls to *glVertex3fv()*.

– PC sampling in Speedshop gives a real−time estimate of the time spent in different sections of the code.

– Basic block counting, from Speedshop, gives an ideal estimate of how much time should be spent, not including memory references.

See the speedshop reference page or the *Speedshop User's Guide* 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 Speedshop.

The CASEVision/WorkShop tools, in particular the performance analyzer, can also help with those measurements. *The WorkShop Overview* introduces the tools.

CPU Tuning: Advanced Techniques

After you have 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 correlation 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" 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.

Modeling to the Graphics Pipeline

The modeling of the database directly affects the rendering performance of the resulting application and therefore has 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.

In OpenGL 1.1, the sprite extension can be used for billboards on certain platforms; see "SGIX_sprite—The Sprite Extension".

Tuning the Geometry Subsystem

The geometry subsystem is the part of the pipeline in which per−polygon operations, such as coordinate transformations, lighting, texture coordinate generation, and clipping are performed. The geometry hardware may also be used for operations that are not strictly transform operations, such as convolution.

This section presents techniques that you can use to tune the geometry subsystem, discussing the following topics:

"Using Peak Performance Primitives for Drawing"

"Using Vertex Arrays"

"Using Display Lists as Appropriate"

"Optimizing Transformations"

"Optimizing Lighting Performance"

"Choosing Modes Wisely"

"Advanced Transform−Limited Tuning Techniques"

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 are 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, see Chapter 16, "System−Specific Tuning."

Use "well−behaved" polygons—convex and planar, with only three or four vertices.

If you use concave and self−intersecting polygons, they are broken down into triangles by OpenGL. For high−quality rendering, you must pass the polygons to GLU to be tessellated. This can make them 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 and minimize state changes to reduce pipeline revalidation.

Using Vertex Arrays

Vertex arrays are available in OpenGL 1.1. They offer the following benefits:

The OpenGL implementation can take advantage of uniform data formats.

The *glInterleavedArrays()* call lets you specify packed vertex data easily. Packed vertex formats are typically faster for OpenGL to process.

The *glDrawArrays()* call reduces subroutine call overhead.

The *glDrawElements()* call reduces subroutine call overhead and also reduces per−vertex calculations because vertices are reused.

Using Display Lists as Appropriate

You can often improve geometry performance by storing frequently−used commands in a display list. If you plan to redraw the same geometry multiple times, or if you have a set of state changes that are applied multiple times, consider using display lists. Display lists allow you to define the geometry or state changes once and execute them multiple times. Some graphics hardware stores display lists in dedicated memory or stores data in an optimized form for rendering (see also "CPU Tuning: Display Lists").

Storing Data Efficiently

Putting some extra effort into generating a more efficient database makes a significant difference when traversing the 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 the data. For peak performance, do as much work as possible before rendering. Preprocessing of data is typically performed at initialization time or when changing from a modeling mode to a fast rendering mode.

Minimizing State Changes

Your program will almost always benefit if you reduce the number of state changes. A good way to do this is to sort your scene data according to what state is set and render primitives with the same state settings together. Primitives should be sorted by the most expensive state settings first. Typically it is expensive to change texture binding, material parameters, fog parameters, texture filter modes, and the lighting model. However, some experimentation will be required to determine which state settings are most expensive on the system you are running on. For example, on systems that accelerate rasterization, it may not be very expensive to disable or enable depth testing or to change rasterization controls such as the depth test function. But if you are running on a system with software rasterization, this may cause the graphics pipeline to be revalidated.

It is also important to avoid redundant state changes. If your data is stored in a hierarchical database, make decisions about which geometry to display and which modes to use at the highest possible level. Decisions that are made too far down the tree can be redundant.

Optimizing Transformations

OpenGL implementations are often able to optimize transform operations if the matrix type is known. Follow these guidelines to achieve optimal transform rates:

Call *glLoadIdentity()* to initialize a matrix rather than loading your own copy of the identity matrix.

Use specific matrix calls such as *glRotate*()*, *glTranslate*()*, and *glScale*()* rather than composing your own rotation, translation, or scale matrices and calling *glLoadMatrix()* or *glMultMatrix()*.

If possible, use single precision such as *glRotatef()*, *glTranslatef()*, and *glScalef()*. (On most systems, this may not be critical because the CPU converts doubles to floats).

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. For some features, the penalties may vary depending on the hardware the application is 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.

- GL_LIGHT_MODEL_LOCAL_VIEWER set to GL_FALSE in *glLightModel()* (the default).
- GL_LIGHT_MODEL_TWO_SIDE set to GL_FALSE in *glLightModel()* (the default).
- Local lights are noticeably more expensive than infinite lights. Avoid lighting where the fourth component of GL_LIGHT_POSITION is nonzero.
- 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.

RGB mode.

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 additional guidelines to achieve peak lighting performance:

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);
/* Draw triangles: */
glBegin(GL_TRIANGLES);
/* Set ambient and diffuse material parameters: */
glColor4f(red, green, blue, alpha);
glVertex3fv(...);glVertex3fv(...);glVertex3fv(...);
glColor4f(red, green, blue, alpha);
glVertex3fv(...);glVertex3fv(...);glVertex3fv(...);
...
glEnd();
```

Disable two−sided lighting unless your application requires it.

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 object.

Disable GL_NORMALIZE.

If possible, provide unit−length normals and don't call*glScale*()* to avoid the overhead of GL_NORMALIZE. On some OpenGL implementations it may be faster to simply rescale the normal, instead of renormalizing it, when the modelview matrix contains a uniform scale matrix.

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.

Choosing Modes Wisely

OpenGL offers many features that create sophisticated effects with excellent performance. For each feature, consider the trade−off between effects, performance, and quality.

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.

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.

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 textures 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).

Eliminate objects or polygons that will be out of sight or too small.

Use fog to increase visual detail without drawing small background objects.

Use culling on a separate processor to eliminate objects or polygons that will be out of sight or too small to see.

Use occlusion culling: draw large objects that are in front first, then read back the depth buffer and use it to avoid drawing objects that are hidden.

Tuning the Raster Subsystem

In the raster system, per−pixel and per−fragment operations take place. The operations include writing color values into the framebuffer or more complex operations like depth buffering, alpha blending, and texture mapping.

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"

"Minimizing Per−Pixel Calculations"

"Using Clear Operations"

"Optimizing Texture Mapping"

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.

Minimizing Per−Pixel Calculations

One way to improve fill−limited drawing is to reduce the work required to render fragments. This section discusses several ways you can do this:

"Avoiding Unnecessary Per−Fragment Operations"

"Using Expensive Per−Fragment Operations Efficiently"

"Using Depth−Buffering Efficiently"

"Balancing Polygon Size and Pixel Operations"

"Other Considerations"

Avoiding 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.

For example, if you are using alpha blending to draw some partially transparent objects, make sure that you disable blending when drawing the opaque objects. Also, if you enable alpha testing to render textures with holes through which the background can be seen, be sure to disable alpha testing when rendering textures or objects with no holes. It also helps to sort primitives so that primitives that require alpha blending or alpha testing to be enabled are drawn at the same time. Finally, you may find it faster to render polygons such as terrain data in back−to−front order.

Organizing Drawing to Minimize Computation

Organizing drawing to minimize per−pixel computation can significantly enhance performance. For example, to minimize depth buffer requirements, disable depth buffering when drawing large background polygons, then draw more complex depth−buffered objects.

Using Expensive Per−Fragment Operations Efficiently

Use expensive per−fragment operations with care. Per−fragment operations, in rough order of increasing cost (with flat−shading being the least expensive and multisampling the most expensive) are as follows:

1. flat−shading

2. Gouraud shading

3. depth buffering
4. alpha blending
5. texturing
6. multisampling

Note: The actual order depends on the system you are running on.

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. Depth buffering is disabled and the sky and ground, then the polygons lying flat on the ground (runway and grid) are drawn without suffering a performance penalty. Then depth buffering is enabled for drawing the mountains and airplanes.

There are 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 is 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)
```

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.

If you are drawing very large polygons such as backgrounds, performance will improve if you use simple fill algorithms. For example, don't set *glShadeModel()* to GL_SMOOTH if smooth shading is not required. Also, disable per−fragment operations such as depth buffering, if possible. If you need to texture the background polygons, consider using GL_REPLACE as the texture environment.

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 with `glDisable(GL_BLEND)`.

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.

Using Clear Operations

When considering clear operations, consider these points:

If possible, avoid clear operations. For example, you can avoid clearing the depth buffer by setting the depth test to GL_ALWAYS.

Avoid clearing the color and depth buffers independently.

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 call:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
```

Be sure to disable dithering before clearing.

Optimizing Texture Mapping

Follow these guidelines when rendering textured objects:

Avoid frequent switching between texture maps. If you have many small textures, consider combining them into a single larger, mosaic 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.

Use texture objects to encapsulate texture data. Place all the *glTexImage*()* calls (including mipmaps) required to completely specify a texture and the associated *glTexParameter*()* calls (which set texture properties) into a texture object and bind this texture object to the rendering context. This allows the implementation to compile the texture into a format that is optimal for rendering and, if the system accelerates texturing, to efficiently manage textures on the graphics adapter.

When using texture objects, call *glAreTexturesResident()* to make sure that all texture objects are resident during rendering. (On systems where texturing is done on the host, *glAreTexturesResident()* always returns GL_TRUE.) If necessary, reduce the size or internal format resolution of your textures until they all fit into memory. If such a reduction creates intolerably fuzzy textured objects, you may give some textures lower priority.

If possible, use *glTexSubImage*D()* to replace all or part of an existing texture image rather than the more costly operations of deleting and creating an entire new image.

Avoid expensive texture filter modes. On some systems, trilinear filtering is much more expensive than nearest or linear filtering.

Tuning the Imaging Pipeline

This section briefly lists some ways in which you can improve pixel processing. Example 15−1 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:

Disable all per−fragment operations.

Define images in the native hardware format so type conversion is not necessary.

For texture download operations, match the internal format of the texture with that on the host.

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. Then process all four images in parallel. Processing four images together is usually faster than processing them individually as single−component images.

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 the color matrix 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 to load texture with an GL_RGB5_A1 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.

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 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.

On most systems, *glCopyPixels()* is faster than *glDrawPixels()*.

Tightly packing data in memory (for example row length=0, alignment=1) is slightly more efficient for host transfer.

Chapter 15
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. The chapter discusses these topics:

"Drawing Pixels Fast"

"Tuning Example"

Drawing Pixels Fast

The code fragment in Example 15−1illustrates how to set an OpenGL state so that subsequent calls to *glDrawPixels()* or *glCopyPixels()* will be fast.

Example 15−1 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 14, "Tuning the Pipeline." Consider a program that draws a lighted sphere, shown in Figure 15−1.

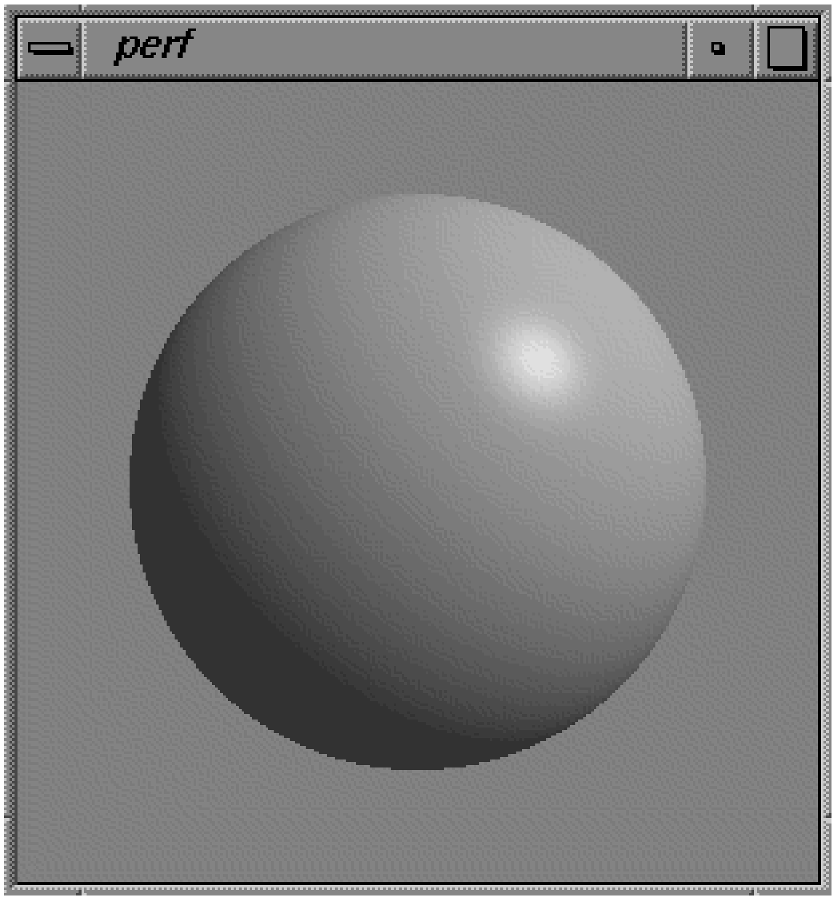


Figure 15−1 Lighted Sphere Created by perf.c

You can use the benchmarking framework in Appendix B, "Benchmarks," for window and timing services. All you have 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 100 MHz Indigo2 Extreme system, this program renders about 0.77 frames per second.

Example 15−2 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; retur
n
 * 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 sing
le
 * 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 seco
nd
 *
 * 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 w
e
         * 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 reac
h
                 * 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 accur
ate
 * 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 * dTo
R);
                      y = sin(latitude * dToR);
                      z = cos(longitude * dToR) * cos(latitude * dTo
R);
                      glNormal3f(x, y, z);
                      glVertex3f(x, y, z);
                      x = sin(longitude * dToR) * cos((latitude+1) *

                                                                 dTo
R);
                      y = sin((latitude+1) * dToR);
                        z = cos(longitude * dToR) * cos((latitude+1)
 *
                                                                 dTo
R);
                      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 windo
w
 * 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.heigh
t);
                                redraw = 1;
                                break;
                        case KeyPress:
                                (void) XLookupString(&event.xkey, bu
f,
                                        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 ProcessEve
nts
*********************************************************************
**/
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 |
                                                      KeyPressMa
sk;
        win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0,
                      300, 300, 0, vi->depth, InputOutput, vi->visu
al,
                      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;

      glColor3f(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) * dTo
R);
                  y = sin((latitude+1) * dToR);
                  z = cos(longitude * dToR) * cos((latitude+1) * dTo
R);
                  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 procedur
e.
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) * dTo
R);
                  y = sin((latitude+1) * dToR);
                  z = cos(longitude * dToR) * cos((latitude+1) * dTo
R);
                  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 is 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 cannot eliminate lighting altogether. The program is already using a fairly simple lighting model (a single infinite light and a nonlocal viewer), so there is 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 is not 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 are not 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, because 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 peak fill rate. Note that you can sometimes improve face culling by performing it in the application; for example, for a sphere you would see just the hemisphere closest to you, and therefore you only have to compute the bounds on latitude and longitude.

Chapter 16

System−Specific Tuning

This chapter first discusses some general issues regarding system−specific tuning, then provides tuning information that is 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, use */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.

Introduction to System−Specific Tuning

Many of the performance tuning techniques discussed in the previous chapters (such as minimizing the number of state changes and disabling features that are not required) are a good idea no matter what system you are running on. Other tuning techniques need to be customized for a particular system. For example, before you sort your database based on state changes, you need to determine which state changes are the most expensive for each system you are interested in running on.

In addition, you may want to modify the behavior of your program depending on which modes are fast. This is especially important for programs that must run at a particular frame rate. To maintain the frame rate on certain systems, you may need to disable some features. For example, if a particular texture mapping environment is slow on one of your target systems, you have to disable texture mapping or change the texture environment whenever your program is running on that platform.

Before you can tune your program for each of the target platforms, you have to do some performance measurements. This is not always straightforward. Often a particular device can accelerate certain features, but not all at the same time. It is therefore important to test the performance for combinations of features that you will be using. For example, a graphics adapter may accelerate texture mapping but only for certain texture parameters and texture environment settings. Even if all texture modes are accelerated, you have to experiment to see how many textures you can use at the same time without causing the adapter to page textures in and out of the local memory.

A more complicated situation arises if the graphics adapter has a shared pool of memory that is allocated to several tasks. For example, the adapter may not have a framebuffer deep enough to contain a depth buffer and a stencil buffer. In this case, the adapter would be able to accelerate both depth buffering and stenciling but not at the same time. Or perhaps, depth buffering and stenciling can both be accelerated but only for certain stencil buffer depths.

Typically, per−platform testing is done at initialization time. You should do some trial runs through your data with different combinations of state settings and calculate the time it takes to render in each case. You may want to save the results in a file so your program does not have to do this test each time it starts up. You can find an example of how to measure the performance of particular OpenGL operations and save the results using the isfast program from the OpenGL web site.

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, 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 stippling 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 operation; 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 on a low end system, 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—except where memory size is critical.

- Use single−precision floats for vertices, normals, and colors.
- Transform paths use single−precision floats—it is 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.

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 generated with *glLineStipple()* 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 on a low−end system. 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 MBps.

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 as follows:

```
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 or GL_DECAL.

– For RGBA textures, make sure the texture environment mode is GL_REPLACE.

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 slow. In particular, avoid using depth−buffered lines. 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 Pixel Operations Effectively

Write your OpenGL program to use the combinations of pixel formats and types listed in Table 16−1, for which the hardware can use DMA. The CPU has to reformat pixels in other format and type combinations.

Table 16−1 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 systems but not on Indy or Indigo^2 XL systems, where the two formats 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 is newly exposed. Using *glCopyPixels()* 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 cannot 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. As a result:

– Rendering while scissoring is turned on is fast.

– Calling *glEnable()* and *glDisable()* with GL_SCISSOR, calling *glScissor()* and pushing and popping attributes that cause a scissor change are slow.

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 (and later).

Note: 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".

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 are 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 O2™ Systems

An O2 system is similar to previous low−end systems in that it divides operations in the OpenGL pipeline between the host CPU and dedicated graphics hardware. However, unlike previous systems, graphics hardware on the O2 handles more of the graphics pipeline in hardware. In particular, it is capable of rasterizing triangle−based primitives directly without the host having to split them into spans, and it performs all of the OpenGL per−fragment operations. The CPU is still responsible for vertex and normal transformation, clipping, lighting, and primitive−level set−up.

In addition to using the CPU for geometry operations and the Graphics Rendering Engine (GRE) for per−fragment operations, a number of imaging extensions and pixel transfer operations are accelerated by the Imaging and Compression Engine (ICE).

Optimizing Geometry Operations

The section "Optimizing Performance on Low−End Graphics Systems"lists recommendations in "Using Geometry Operations Effectively". Many of these recommendations apply to the O2 system as well. There are, however, some differences worth mentioning:

Generic 3D transformations with perspective are comparable in speed to 2D transformations because the floating−point pipeline in the R5000 and R10000 CPUs is much faster than previous−generation CPUs. However, always put perspective in the projection matrix and not in the modelview matrix to allow for faster normal transformation.

Minimize attribute setup; attribute setup for each primitive is performed on the CPU. For example:

– Use flat shading if the color of the model changes rarely or not within the same primitives that make up the model.

– Don’t enable depth−buffering when rendering lines.

– Turn off polygon offset when not in use.

– Choose a visual with no destination alpha planes if destination alpha blending is not used.

When using fog, set the *param* argument to GL_LINEAR instead of GL_EXP or GL_EXP2. GL_LINEAR uses vertex fogging, which is hardware accelerated on O2 systems, instead of per−pixel fogging, which is not.

When continuously rendering a large amount of static geometry elements, consider storing the geometry elements in display lists. When vertices and vertex attributes are stored in display lists, the R10000 CPU can prefetch the data in anticipation of its use and thus reduce read latency for data that cannot fit in the caches.

The n32 version of the OpenGL is somewhat faster than the o32 version due to the more efficient parameter passing convention and the larger number of floating− point registers that the n32 compilation mode offers. Furthermore, using n32 can improve application speed because the compiler can generate MIPS IV code and schedule instructions optimally for the R5000 or the R10000 CPU.

Lighting on O2 systems is faster than on previous low−end systems because of the better floating−point performance of its CPUs. However, the larger the number or the more complex the lights (local lights, for instance), the larger the amount of work the CPU has to perform. Two−sided lighting is not a "free" operation, so consider using single−sided lighting, if possible.

Optimizing Line Drawing

Line drawing for low−end systems is discussed in some detail in"Optimizing Line Drawing". On O2 systems, almost all line rendering (rasterization) modes are hardware supported.

The following kinds of lines need to be rasterized by the CPU and will perform significantly *slower*:

- anti−aliased RGB lines that are either wide (line width greater than 1) or stippled
- all types of anti−aliased color−index lines

Optimizing Triangle Drawing

Triangle drawing for low−end systems is discussed in some detail in"Optimizing Triangles and Polygons". Note the following points:

- Triangle strips are the most optimal triangle path through the OpenGL pipeline. Maximize the number of vertices between *glBegin()* and *glEnd()*.
- Polygon stippling is not hardware supported. Because a stippled polygon has to be rasterized on the CPU, enabling polygon stippling will cause a significant performance degradation.

  If the application is using polygon stippling for screen−door transparency effects, consider instead using alpha blending to emulate transparency. If using alpha blending is not possible, consider setting the GLCRMSTIPPLEOPT environment variable. Setting this variable enables an optimization that uses the stencil planes to emulate polygon stippling if the application does not use the stencil planes. However, note that if the stipple pattern changes often during the rendering of a frame, the performance benefits may be lost to the time spent repainting the stencil planes with the different patterns.

Using Per−Fragment Operations Effectively

This section discusses how to use per−fragment operations effectively on O2 systems.

Getting Optimum Fill Rates

The rasterization hardware has the same fill rates whether the shading model is smooth or flat. If the application is rendering very large areas, there should be little difference in the performance between smooth and flat shading. However, remember that setting up smooth−shaded primitives is more expensive on the CPU side.

Framebuffer Configurations

The framebuffer on O2 systems can be configured four different ways (16, 16+16, 32, 32+32) to allow applications to trade off memory usage and rendering speed for image quality. Apart from pixel depth, the other main difference between these framebuffer types is where the back−buffer pixels of a double−buffered visual reside. For the 16− and 32−bit framebuffer, the front and back buffers share the same pixel with each buffer taking half of the pixel. For the 16+16 and 32+32 framebuffers, the back buffer is allocated as needed in a different region from the main framebuffer. As a result, 16+16 and 32+32 buffers can have visuals with the same color depth for single−buffered and double−buffered visuals but will need more memory in that case.

The framebuffer’s configuration (size and depth) affects fill rate performance. In general, the deeper the framebuffer, the more data the GRE (graphics rendering engine) needs to write to memory to update the framebuffer and the more data the graphics back−end needs to read from the framebuffer to refresh the screen. Note that for double−buffered applications, better fill rates can be achieved with the split 16+16 framebuffer than with the 32−bit framebuffer. This is because the new color information can be written to the pixels directly instead of having to be combined with what is in the framebuffer. This is especially important for fill−rate limited texture mapping operations, buffer clears and pixel DMAs.

Texture Mapping

An O2 system stores texture maps in system memory. The amount of texture storage is therefore limited only by the amount of physical memory on the system. Texture memory is partitioned into 64 KB tiles. Texture memory is made available on a tile basis; the actual memory usage for a texture is rounded up to 64 KB and the memory usage will be higher than if the same texture were to be packed optimally in memory.

Tile−based texture memory also means that the minimum memory usage for any texture is one tile and the amount of "wasted" texture memory can quickly add up if the application uses a large number of small textures. In that case, consider combining small textures into larger ones and using different texture coordinates to access different sections of the larger texture map.

The following texture formats are supported directly by the graphics hardware and require no conversion when specified by the application:

- 8 bit luminance or intensity
- 16 bit luminance−alpha (8:8 format)
- 16−bit RGBA (5:5:5:1 format)
- 16 bit RGBA (4:4:4:4 format)
- 32 bit RGBA (8:8:8:8 format)

Applications that use more than one texture should use texture objects, now part of OpenGL 1.1, for faster switching between multiple textures. Although fast, binding a texture is not a free operation and judicious minimization of its use during frame rendering will increase performance. This can be achieved by rendering all the primitives that use the same texture object at the same time.

The texture filters GL_NEAREST and GL_LINEAR result in the best texture fill rates, whereas GL_LINEAR_MIPMAP_LINEAR results in the worst fill rate. In cases where texture maps are being minified and only bilinear filtering is required, consider using mipmaps with the minification filter set to GL_LINEAR_MIPMAP_NEAREST. This filter gives the graphics engine better cache locality and better fill rates.

The 3D texture mapping, texture color table, and texture scale bias extensions are supported by the O2 OpenGL implementation, but are not hardware accelerated. Enabling one of these extensions will therefore result in significantly slower rendering.

Front and Back Buffer Rendering

The graphics rendering engine does not allow updating both the front and back buffers at the same time (`glDrawBuffer(GL_FRONT_AND_BACK)`). In order to support this functionality, the OpenGL needs to specify the primitive being rendered to the graphics hardware twice, once for both the front and back buffer. This is an expensive operation and applications should try to avoid using concurrent updates to both front and back buffers.

Getting Optimum Pixel DMA Rates

The following is a table of pixel types and formats for which hardware DMA can be used.

Table 16−2 Pixel Formats and Types That Work With DMA on O2 Systems

Format	Type
GL_COLOR_INDEX	GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT
GL_STENCIL_INDEX	GL_UNSIGNED_INT
GL_DEPTH_COMPONENT	GL_UNSIGNED_INT
GL_RGB	GL_UNSIGNED_BYTE, GL_UNSIGNED_BYTE_3_3_2_EXT
GL_RGBA	GL_UNSIGNED_BYTE, GL_UNSIGNED_INT_8_8_8_8_EXT
GL_ABGR_EXT	GL_UNSIGNED_BYTE, GL_UNSIGNED_INT_8_8_8_8_EXT
GL_LUMINANCE	GL_UNSIGNED_BYTE
GL_YCRCB_422_SGIX	GL_UNSIGNED_BYTE

The pixel DMA paths support stencil, depth, and alpha tests, fogging, blending, and texturing.

Stencil indices can be sent via DMA as 32−bit unsigned int values, where the most significant 8 bits are transferred, using a stencil shift value of -24 for draw operations and 24 for read operations.

Depth components can be sent via DMA as 24−bit unsigned int values, using a depth scale of 256 for draw operations and 1/256.0 for read operations. For draw operations, the depth test must be enabled with a function of GL_ALWAYS, and the color buffer must be set to GL_NONE.

Pixel zooms are accelerated for whole integer factors from −16 to 16, and integer fractions from −1/16 to 1/16 on all DMA paths.

Most read pixel operations on O2 will be significantly faster when the destination buffer and row lengths are 32−byte aligned.

Imaging Pipeline Using ICE

O2 systems contain a multi−purpose compute ASIC called the Imaging and Compression Engine (ICE) which serves both the needs of DCT−based compression algorithms and of OpenGL image processing. All the elements in the OpenGL imaging pipeline (that is the pixel transfer modes) are implemented on ICE, but some functions (such as convolution and color matrix multiplication) benefit a lot while others (like histogram and color table) don't benefit as much. This section discusses the support provided by ICE and gives some programming tips.

Pixel Formats. ICE supports the 8−bit GL_RGBA, GL_RGB, and GL_LUMINANCE pixel formats. Because the O2 graphics hardware does not support an RGB framebuffer type, RGB pixels have to be converted to RGBA before they can be displayed. Instead of using the CPU to perform this conversion, *glDrawPixels()* uses the wide loads and stores and DMA engine on ICE. It is possible to use other pixel formats such as luminance−alpha or color index, but for those formats, the CPU performs all image processing calculations.

64 KB Tiles. The memory system natively allocates memory for the framebuffer and pbuffers in 64 KB tiles. ICE takes advantage of this by having a translation look−aside buffer (TLB) in the DMA engine that maps 64 KB tiles.

Buffer to buffer fast path. Because ICE can directly map tiles without further manipulation, it is fastest to go buffer to buffer (i.e. *glCopyPixels()*) for the imaging pipeline on O2. While not explicitly an imaging operation, ICE supports span conversion between GL_RGBA and GL_LUMINANCE on the pixel transfer path including *glCopyPixels()*. *glDrawPixel()* is the next fastest path.

Image Width. Any image width up to 2048 pixels is permitted, but image widths that are modulo 16 pixels are optimal. If the image is not modulo 16, the CPU uses *bcopy()*; to pad the image to closest modulo 16 width. Note that setting row pack, row unpack, and certain clipping and zoom combinations can cause the internal image width to change from what was modulo 16 pixels.

Number Formats. The vector processor on ICE dictates to a large extent the numerical representation of coefficients that can be used. There are two number formats on ICE: integer and fixed point (s.15). Therefore values should be either [−1.0, 1.0) or strictly integer. Numbers outside this range force the library to perform the calculations on the CPU. Developers have not found this to be too restricting as a multiplication; by 1.9, for example, can also be expressed as a multiplication of 0.95 followed by a multiplication of 2.0. OpenGL allows this trick through use of the post color scale functions.

Memory. Some programming restrictions arise from the need to balance the amount of state kept on the chip and the amount of memory available for image data.   The 6 KB of data RAM is organized into 3 banks. Bank C is 2 KB and is used for storing color tables, histogram, convolution coefficients, and 256 bytes of internal state. In order to remain on the fast path, the total bytes used for items in Bank C must be less than 2 KB. Because of that limitation, two color tables specified as GLbyte and GL_RGBA will not be hardware accelerated. This is not a problem if the application can specify the color tables as GL_LUMINANCE or GL_LUMINANCE_ALPHA.

Color Tables. The number, type, and format of color tables is important to keep the application on the fast path. Up to two color tables or one color table and one histogram can be accelerated on the O2 imaging fast path. The internal format of the color table can be luminance, luminance−alpha, or rgba. The color table type must be GL_BYTE. While the texture color table is not supported, using the color table extension on texture load is an alternative.

Convolution. Both general and separable convolutions are hardware accelerated on O2. Convolution kernel sizes that are accelerated are 3x3, 5x5, and 7x7. Applications can gain further performance improvement by specifying the kernel as GL_INTENSITY (note that this is different than GL_LUMINANCE). O2 systems cannot accelerate convolutions and histograms at the same time. See "EXT_convolution—The Convolution Extension" for more information.

Separating Components. On other graphics architectures, there is a significant advantage to processing image components one at a time. Some OpenGL implementations use the color matrix multiply function to separate out components. There is no advantage to separating out a component on O2 by using the color matrix multiply function. The intent was to use the matrix multiply for color correction. Unlike the color scale and bias and convolution, matrix multiply values should be in the [−1.0, 1.0) range for hardware acceleration.

Histograms. Histograms are internal calculated with 16−bit bins, and the internal format is only GL_RGBA. While an application can ask for different formats, the histogram is always calculated as RGBA.

Extensions Supported by O2 Systems

O2 systems currently support the following extensions:

Pixel Extensions: EXT_abgr, EXT_packed_pixels, SGIX_interlace

Blending Extensions: EXT_blend_color, EXT_blend_logic_op, EXT_blend_minmax, EXT_blend_subtract

Imaging extensions:. EXT_convolution, EXT_histogram, SGI_color_matrix, SGI_color_table

Buffer and Pbuffer extensions: EXT_import_context, EXT_visual_info, EXT_visual_rating, SGIX_dm_pbuffer, SGIX_fbconfig, SGIX_pbuffer

Texture extensions: EXT_texture3D, SGIS_texture_border_clamp, SGIS_texture_color_table, SGIS_texture_edge_clamp,

Supported only on O2 systems: SGIS_generate_mipmap, SGIS_texture_scale_bias. These two extensions are not discussed in this manual.

Video and swap control extensions: SGI_swap_control, SGI_video_sync, SGIX_video_source.

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 Indigo2 IMPACT systems, see "Optimizing Performance on Indigo2 IMPACT and OCTANE Systems".

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 ABGR extension is hardware accelerated (see "EXT_abgr—The ABGR Extension").

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 "EXT_abgr—The ABGR Extension".

The combinations of types and formats shown in Table 16−3are the fastest.

Table 16−3 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 Indigo^2 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 Indigo2 IMPACT and OCTANE

Systems

This section provides performance tips for Indigo2 IMPACT and OCTANE graphics systems. All information applies to all Indigo2 IMPACT and OCTANE systems, except sections on texture mapping, which do not apply to the Indigo2 Solid IMPACT and the OCTANE SI (without hardware texture mapping). You learn about these topics

"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, 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". Note in this context that texture−mapping is greatly accelerated on Indigo2 IMPACT and OCTANE systems with hardware texture−mapping.

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".

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 Indigo2 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 (width greater than 1.0) 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

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 independent 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

Texturing capabilities of the Indigo2 IMPACT and OCTANE systems differ, as shown in the following table:

Table 16−4 Texturing on Indigo2 and OCTANE Systems

Platform	Supported Texturing
Indigo2 Solid IMPACT	Software texturing
OCTANE SI	Software texturing
Indigo2 High IMPACT	Hardware texturing
OCTANE SI with hardware textures	Hardware texturing
Indigo2 Maximum IMPACT	Hardware texturing
OCTANE MXI	Hardware texturing

Texture−mapping is greatly accelerated on systems with hardware texture, and is only slightly slower than non−textured fill rates. It also 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 texture objects to keep as many textures resident in texture memory as possible. You can 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.

Texture objects are part of OpenGL 1.1. For OpenGL 1.0, they were implemented as the texture object extension (EXT_texture_object).

Use the texture−LOD extension to clamp LOD values, which has the side effect of communicating to the system which mipmap levels it needs to keep resident in texture memory. For more information, see "SGIS_texture_lod—The Texture LOD Extension".

Use subtextures to make texture definitions 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 *glTexSubImage2D()* to redefine the image as needed.

Subtextures are part of OpenGL 1.1. They were implemented as the subtexture extension (EXT_subtexture) in OpenGL 1.0.

Use the GL_RGBA4 internal format to improve performance and conserve memory. 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 use less space.

Internal formats are part of OpenGL 1.1. They were implemented as part of the texture extension in OpenGL 1.0.

Use the GL_RGBA4 internal format and the packed pixels extension to minimize disk space and improve download rate (see "EXT_packed_pixels—The Packed Pixels Extension").

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 "EXT_texture3D—The 3D Texture Extension".

If you use GL_LUMINANCE and GL_LUMINANCE_ALPHA textures, you can speed up loading by using the texture−select extension (see"SGIS_texture_select—The Texture Select Extension").

For Indigo2 IMPACT graphics, data coherence enhances performance. For example:

– When you draw your geometry, cluster points, short lines, or very small triangles so that you are not jumping around the texture (you want to maintain texture data coherency).

– If any minification is done to the texture, mipmaps result in improved performance.

– When you use the pixel texture extension, performance varies based on the coherency of the lookup of pixel color data as texture coordinates. Applications have no control over this.

Using Images

This section provides some tips for using images on Indigo2 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 Indigo2 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 Indigo2 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 *glPixelMap()*, use the Silicon Graphics color table extension, discussed in "SGI_color_table—The Color Table Extension", 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. Indigo2 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, Indigo2 IMPACT hardware postpones the expansion, which speeds up processing.

The convolution extension, discussed in "EXT_convolution—The Convolution Extension" has been optimized. If possible, use the extension with separable convolution filters.

Indigo2 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 Indigo2 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 color table and the histogram compete for limited resources with other OpenGL applications.

Accelerating Color Space Conversion

Indigo2 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 "SGI_color_matrix—The Color Matrix Extension" 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 Indigo2 IMPACT systems:

When the display list is compiled, most OpenGL functions are stored in a format that the hardware can 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 Indigo2 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 are working with, whether it is embedded in other lists, and other factors.

Display List Compilation on Indigo2 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

glHint(), *glClear()*, and *glScissor()*

glEnable() and *glDisable()*

glPushAttrib() and *glPopAttrib()*

all evaluator functions

most OpenGL extensions

DMA Display Lists on Indigo2 IMPACT Systems

If a display list meets certain criteria, Indigo2 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 "SGIX_pbuffer—The Pixel Buffer Extension".

Optimizing Performance on RealityEngine Systems

This section provides information on optimizing applications for RealityEngine and RealityEngine2. It discusses these topics:

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 command (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. Internal formats are part of OpenGL 1.1; they were part of the texture extension in OpenGL 1.0.

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 "SGIS_sharpen_texture—The Sharpen Texture Extension" and "SGIS_detail_texture—The Detail Texture Extension".

If textures are changing frequently, use subtextures to incrementally load texture data. RealityEngine systems are optimized for 32 x 32 subimages.

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 internal format, which is part of OpenGL 1.1 and part of the texture extension for OpenGL 1.0).

Local lighting or multiple lights are possible without an unacceptable degradation in performance. As you turn on more lights, performance degrades slowly.

Simultaneous clearing of depth and color buffers is optimized in hardware.

Antialiased lines and points are hardware accelerated.

Optimizing Use of the Vertex Arrays

Vertex arrays were implemented as an extension to OpenGL 1.0 and are part of OpenGL 1.1. If you use vertex arrays, 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:

glVertex2f

glVertex3f

glNormal3f glVertex3f

glColor3f glVertex3f

glColor4f glVertex3f

glNormal3f glVertex3f

glTexCoord2f glVertex3f

glColor4f glTexCoord2f glVertex3f

glColor3f glNormal3f glVertex3f

glColor4f glNormal3f glVertex3f

glNormal3f glTexCoord2f glVertex3f

glColor4f glTexCoord2f glNormal3f glVertex3f

Optimizing Multisampling and Transparency

Multisampling provides full−scene antialiasing with performance sufficient for a real−time visual simulation application. However, it is not 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 are not 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 is not 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 operation. 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 operation requires that every pixel in the window be rendered every frame; pixels that are not touched remain unchanged. This clear operation 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. Eight 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).

Polygon offsets are supported in OpenGL 1.1 and were part of the Polygon Offset extension in OpenGL 1.0.

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, because more buffer memory may need to be accessed. For more information about screen−door transparency, see "SGIS_multisample—The Multisample Extension".

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 "SGIX_fbconfig—The Framebuffer Configuration Extension"). 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") 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.

The following set of circumstances has 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.

Optimizing Performance on InfiniteReality Systems

This section discusses optimizing performance on InfiniteReality systems in the following sections:

"Managing Textures on InfiniteReality Systems"

"Offscreen Rendering and Framebuffer Management"

"Optimizing State Changes"

"Miscellaneous Performance Hints"

Managing Textures on InfiniteReality Systems

The following texture management strategies are recommended for InfiniteReality systems:

Using the texture_object extension (OpenGL 1.0) or texture objects (OpenGL 1.1) usually yields better performance than using display lists.

Note that on RealityEngine systems, using display lists was recommended. On InfiniteReality systems, using texture objects is preferred.

OpenGL will make a copy of your texture if needed for context switching, so deallocate your own copy as soon as possible after loading it. Note that this behavior differs from RealityEngine behavior.

Note that RealityEngine and InfiniteReality systems differ here:

- On RealityEngine systems, there is one copy of the texture on the host, one on the graphics pipeline. If you run out of texture memory, OpenGL sends the copy from the host to the graphics pipeline after appropriate cleanup.

– On Infinite Reality systems, only the copy on the graphics pipe exists. If you run out of texture memory, OpenGL has to save the texture that didn't fit from the graphics pipe to the host, then clean up texture memory, then reload the texture. To avoid these multiple moves of the texture, be sure to always clean up textures you no longer need so you don't run out of texture memory.

  This approach has the advantage of very fast texture loading because no host copy is made.

To load a texture immediately, follow this sequence of steps:

1. Enable texturing.
2. Bind your texture.
3. Call *glTexImage*()*.

To define a texture without loading it into the hardware until the first time it is referenced, follow this sequence of steps:

1. Disable texturing.
2. Bind your texture.
3. Call *glTexImage*()*.

Note that in this case, a copy of your texture is placed in main memory.

Don't overflow texture memory, or texture swapping will occur.

If you want to implement your own texture memory management policy, use subtexture loading. You have two options. For both options, it is important that after initial setup, you never create and destroy textures but reuse existing ones:

– Allocate one large empty texture, then call *glTexSubImage*()* to load it piecewise, and use the texture matrix to select the relevant portion.

– Allocate several textures, then fill them in by calling *glTexSubImage*()* as appropriate.

Use 16−bit texels whenever possible; RGBA4 can be twice as fast as RGBA8. As a rule, remember that bigger formats are slower.

If you need a fine color ramp, start with 16−bit texels, then use a texture lookup table and texture scale/bias.

Texture subimages should be multiples of 8 texels wide for maximum performance.

For loading textures, use pixel formats on the host that match texel formats on the graphics system.

Avoid OpenGL texture borders; they consume large amounts of texture memory. For clamping, use the GL_CLAMP_TO_EDGE_SGIS style defined by the SGIS_texture_edge_clamp extension (see "SGIS_texture_edge/border_clamp—Texture Clamp Extensions"). This extension is identical to the old IRIS GL clamping semantics on RealityEngine.

Offscreen Rendering and Framebuffer Management

InfiniteReality systems support offscreen rendering through a combination of extensions to GLX:

pbuffers are offscreen pixel arrays that behave much like windows, except that they're invisible. See "SGIX_pbuffer—The Pixel Buffer Extension".

fbconfigs (framebuffer configurations) define color buffer depths, determine presence of Z buffers, and so on. See "SGIX_fbconfig—The Framebuffer Configuration Extension".

glXMakeCurrentReadSGI() allows you to read from one window or pbuffer while writing to another. See "EXT_make_current_read—The Make Current Read Extension".

In addition, *glCopyTexImage*()* allows you to copy from pbuffer or window to texture memory. This function is supported through an extension in OpenGL 1.0 but is part of OpenGL 1.1.

For framebuffer memory management, consider the following tips:

Use pbuffers. pbuffers are allocated by "layer" in unused portions of the framebuffer.

If you have deep windows, such as multisampled or quad− buffered windows, then you'll have less space in the framebuffer for pbuffers.

pbuffers are swappable (to avoid collisions with windows), but not completely virtualized, that is, there is a limit to the number of pbuffers you can allocate. The sum of all allocated pbuffer space cannot exceed the size of the framebuffer.

pbuffers can be volatile (subject to destruction by window operations) or nonvolatile (swapped to main memory in order to avoid destruction). Volatile pbuffers are recommended because swapping is slow. Treat volatile pbuffers like they were windows, subject to exposure events.

Optimizing State Changes

As a rule, it is more efficient to change state when the relevant function is disabled than when it is enabled. For example, when changing line width for antialiased lines, call

```
glLineWidth(width);
glEnable(GL_LINE_SMOOTH);
```

As a result of this call, the line filter table is computed just once, when line antialiasing is enabled. If you call

```
glEnable(GL_LINE_SMOOTH);
glLineWidth(width);
```

the table may be computed twice: Once when antialiasing is enabled, and again when the line width is changed. As a result, it may be best to disable a feature if you plan to change state, then enable it after the change.

The following mode changes are fast: sample mask, logic op, depth function, alpha function, stencil modes, shade model, cullface, texture environment, matrix transforms.

The following mode changes are slow: texture binding, matrix mode, lighting, point size, line width.

For best results, map the near clipping plane to 0.0 and the far clipping plane to 1.0 (this is the default). Note that a different mapping, for example 0.0 and 0.9, will still yield good result. A reverse mapping, such as near = 1.0 and far = 0.0, noticeably decreases depth−buffer precision.

When using a visual with a 1−bit stencil, it is faster to clear both the depth buffer and stencil buffer than it is to clear the depth buffer alone.

Use the color matrix extension for swapping and smearing color channels. The implementation is optimized for cases in which the matrix is composed of zeros and ones.

Be sure to check for the usual things: indirect contexts, drawing images with depth buffering enabled, and so on.

Triangle strips that are multiples of 10 (12 vertices) are best.

InfiniteReality systems optimize 1−component pixel draw operations. They are also faster when the pixel host format matches the destination format.

Bitmaps have high setup overhead. Consider these approaches:

– If possible, draw text using textured polygons. Put the entire font in a texture and use texture coordinates to select letters.
– To use bitmaps efficiently, compile them into display lists. Consider combining more than one into a single bitmap to save overhead.
– Avoid drawing bitmaps with invalid raster positions. Pixels are eliminated late in the pipeline and drawing to an invalid position is almost as expensive as drawing to a valid position.

Miscellaneous Performance Hints

Minimize the amount of data sent to the pipeline.

– Use display lists as a cache for geometry. Using display lists is critical on Onyx 1 system. It is less critical, but still recommended, on Onyx2 systems. The two systems performance differs because the bus between the host and the graphics is faster on Onyx2 systems.

  The display list priority extension (see "SGIX_list_priority—The List Priority Extension") can be used to manage display list memory efficiently.
– Use texture memory or offscreen framebuffer memory (pbuffers) as a cache for pixels.
– Use small data types, aligned, for immediate−mode drawing such as RGBA color packed into a 32−bit word, surface normals packed as three shorts, texture coordinates packed as two shorts). Smaller data types mean, in effect, less data to transfer.
– Use the packed vertex array extension.

Render with exactly one thread per pipe.

Use multiple OpenGL rendering contexts sparingly. The rendering context−switching rate is about 60,000 calls per second, assuming no texture swapping, so each call to *glXMakeCurrent()* costs the equivalent of 100 textured triangles or 800 32−bit pixels.

Appendix A
OpenGL and IRIS GL

The first step in porting an 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. For additional information, see 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".

blendfunction

See "Blending Extensions".

convolve

See "EXT_convolution—The Convolution Extension".

displacepolygon

The OpenGL equivalent, *glPolygonOffset()*, is more general than *displacepolygon()*. You may need to tweak the parameter values to get the proper results. See "Polygon Offset" starting on page 247 of the *OpenGL Programming Guide, Version 1.1*.

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

Used to be supported through an extension in OpenGL 1.0. For OpenGL 1.1, see the glTexSubImage1D and glTexSubImage2D reference pages and "Replacing All or Part of a Texture Image" starting on page 332 of the *OpenGL Programming Guide, Version 1.1*.

gamma

Use the XSGIvc extension. See "Stereo Rendering".

glcompat GLC_SET_VSYNC, GLC_GET_VSYNC, GLC_VSYNC_SLEEP

For GLC_GET_VSYNC, use *glXGetVideoSyncSGI()*. For GLC_VSYNC_SLEEP, use *glXWaitVideoSyncSGI()*. See "SGI_swap_control—The Swap Control Extension".

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(lrect
read, glTexImage(texture)
```

Use *glGetHistogramEXT()* and *glHistogramEXT()*; see "EXT_histogram—The Histogram and Minmax Extensions".

ilbuffer, ildraw, readsource(SRC_ILBUFFER)

This function provides accelerated drawing to offscreen framebuffer memory.

See "SGIX_pbuffer—The Pixel Buffer Extension".

istexloaded

Use *glAreTexturesResident()*; see the glAreTexturesResident reference page or "A Working Set of Resident Textures" starting on page 351 of the *OpenGL Programming Guide, Version 1.1*.

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. Some of the functionality is available in the sprite extension; see "SGIX_sprite—The Sprite Extension".

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 "EXT_histogram—The Histogram and Minmax Extensions".

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 co
mpletion 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 "SGIS_multisample—The Multisample Extension".

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(lrect read), glTexImage(texdef),`

On RealityEngine systems, pixel mapping 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 " multisample, getmultisample, msalpha, msmask, mspattern, mssize (multisample antialiasing)".

polymode

OpenGL doesn’t support PYM_HOLLOW. *glPolygonMode(GL_LINE)* is the closest approximation. See also the glPolygonOffset reference page and "Polygon Offset" starting on page 247 of the *OpenGL Programming Guide, Version 1.1*.

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".

See "SGI_color_matrix—The Color Matrix Extension" 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 glTexSubImage1D and glTexSubImage2D reference pages and "Replacing All or Part of a Texture Image" on page 332ff of the *OpenGL Programming Guide, Version 1.1*.

tevdef, tevbind

TV_COMPONENT_SELECT (the ability to pack multiple shallow textures together, then unpack and select one of them during drawing) is supported on IMPACT and InfiniteReality systems via the texture select extension (see "SGIS_texture_select—The Texture Select Extension".

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 *glBindTexture()*; see the glBindTexture reference page and "Creating and Using Texture Objects" on page 348ff of the *OpenGL Programming Guide, Version 1.1*.

Detail texturing (TX_TEXTURE_DETAIL) is supported. Use *glDetailTexFuncSGIS()*; see "SGIS_detail_texture—The Detail Texture Extension".

Simple texture memory management (TX_TEXTURE_IDLE) is supported. Use *glPrioritizeTextures()*; see the glPrioritizeTextures reference page "A Working Set of Resident Textures" starting on page 351 of the *OpenGL Programming Guide, Version 1.1*.

texdef

1D, 2D, and 3D textures are supported. See *glTexImage1D()*, *glTexImage2D()*, and *glTexImage3DEXT()*; see "EXT_texture3D—The 3D Texture Extension".

TX_FAST_DEFINE is not supported. Loading subtextures is still possible, however; use *glTexSubImage2D()*. See the glTexSubImage1D and glTexSubImage2D reference pages and "Replacing All or Part of a Texture Image" starting on page 332 of the *OpenGL Programming Guide, Version 1.1*.

The TX_BILINEAR_LEQUAL and TX_BILINEAR_GEQUAL filtering options, which are used to implement shadows, are not supported. On InfiniteReality systems, the shadow extension is supported; see "SGIX_shadow, SGIX_depth_texture, and SGIX_shadow_ambient—The Shadow Extensions".

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. On InfiniteReality systems, mipmapping is supported.

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 "SGIS_sharpen_texture—The Sharpen Texture Extension".

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 "SGIS_detail_texture—The Detail Texture Extension".

The TX_WRAP mode TX_SELECT is supported by the texture select extension to OpenGL 1.1. See "SGIS_texture_select—The Texture Select Extension". 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.

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 "SGI_texture_color_table—The Texture Color Table Extension". Use *glColorTableSGI()* for information about loading the lookup table; see "SGI_color_table—The Color Table Extension".

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
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.

Appendix B

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 15, "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; retur
n
 * 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 sing
le
 * 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 w
e
         * 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 reac
h
                 * 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 accur
ate
 * 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 windo
w
 * 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.wid
th,
                                        WindowHeight =
                                              event.xconfigure.heigh
t);
                                redraw = 1;
                                break;
                        case KeyPress:
                                (void) XLookupString(&event.xkey, bu
f,
                                        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 ProcessEve
nts
********************************************************************
**/
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),VisualAttribute
s);
        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, AllocNon
e);

        /* Create a window: */
        swa.border_pixel = 0;
        swa.event_mask = ExposureMask | StructureNotifyMask |
                                                          KeyPressMa
sk;
```

```
        win = XCreateWindow(dpy, RootWindow(dpy, vi−>screen), 0, 0,
                        300, 300, 0,vi−>depth, InputOutput, vi−>visu
al,
                        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);
        }
```

Appendix C
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* rat
e)
```

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 initialize libisfast by calling

```
int IsFastXOpenDisplay(const char* displayName);
```

Perform cleanup by calling

```
void IsFastXCloseDisplay(void);
```

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.

Appendix D

Extensions on Different Silicon Graphics Systems

This appendix lists all extensions supported for InfiniteReality systems, OCTANE and Indigo2 IMPACT systems, and O2 systems. Note that while the list is comprehensive, this guide only discusses those extensions that are either available or scheduled to be available on more than one platform.

Table D−1 Extension on Different Silicon Graphics Systems

Extension	InfiniteReality	OCTANE and IMPACT	O2
EXT_abgr	X	X	X
EXT_blend_color	X	X	X
EXT_blend_logic_op	X	X	X
EXT_blend_minmax	X	X	X
EXT_blend_subtract	X	X	X
EXT_convolution	X	X	X
EXT_histogram	X	X	X
EXT_packed_pixels	X	X	X
EXT_texture_3D	X	X	X
SGI_color_matrix	X	X	X
SGI_color_table	X	X	X
SGI_texture_color_table	X	X	X
SGIS_detail_texture	X	X	
SGIS_fog_function	X		
SGIS_multisample	X		
SGIS_point_line_texgen	X		
SGIS_point_parameters	X		
SGIS_sharpen_texture	X		
SGIS_texture_border_clamp		X	
SGIS_texture_edge_clamp	X		X
SGIS_texture_filter4	X	X	
SGIS_texture_LOD	X	X	
SGIS_texture_select	X	X	
SGIX_calligraphic_fragment	X		
SGIX_clipmap	X		
SGIX_fog_offset	X		
SGIX_instruments	X		
SGIX_interlace	X		X
SGIX_ir_instrument1	X		
SGIX_flush_raster	X		
SGIX_list_priority	X		
SGIX_reference_plane	X		
SGIX_shadow	X		
SGIX_shadow_ambient	X		
SGIX_sprite	X		
SGIX_texture_add_env	X		
SGIX_texture_lod_bias	X		
SGIX_texture_scale_bias	X		X
SGIX_depth_texture	X		

Index