OpenGL® on Silicon Graphics Systems

007-2392-003

CONTRIBUTORS

Written by Renate Kempf and Jed Hartman. Revised by Ken Jones.

Illustrated by Dany Galgani, Martha Levine, and Chrystie Danzer

Production by Allen Clardy and Karen Jacobson

Engineering contributions by Allen Akin, Steve Anderson, David Blythe, Sharon Rose Clay, Terrence Crane, Kathleen Danielson, Tom Davis, Celeste Fowler, Ziv Gigus, David Gorgen, Paul Hansen, Paul Ho, Simon Hui, George Kyriazis, Mark Kilgard, Phil Lacroute, Jon Leech, Mark Peercy, Dave Shreiner, Chris Tanner, Joel Tesler, Gianpaolo Tommasi, Bill Torzewski, Bill Wehner, Nancy Cam Winget, Paula Womack, David Yu, and others.

Some of the material in this book is from "OpenGL from the EXTensions to the SOLutions," which is part of the developer's toolbox.

St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

New Features in This Guide

In addition to miscellaneous changes throughout, this revision includes the following changes:

General Changes

The guide now reflects OpenGL 1.3, GLX 1.3, and GLU 1.3 and current Silicon Graphics visualization systems. Many of the changes reflect support for Silicon Graphics Onyx4 UltimateVision systems on IRIX and Silicon Graphics Prism systems on Linux.

New Chapters

- Chapter 7, “Vertex Processing Extensions”
- Chapter 13, “Vertex and Fragment Program Extensions”

Extensions Deprecated

The functionality of the following extensions is now integrated into OpenGL, GLX, and GLU but the extensions remain in this guide for reference by developers using older Silicon Graphics systems—such as VPro, InfinitePerformance, and InfiniteReality:

Resource control extensions	Make current read, framebuffer configuration, and pixel buffer
Texturing extensions	Texture objects, subtexture, copy texture, 3D texture, texture edge/border clamp, texture LOD, texture environment add, and texture LOD bias
Rendering extensions	Blending extensions, multisample, point parameters, shadow, and depth texture
Imaging extensions	Blend logic op, convolution, histogram and minmax, packed pixels, color matrix, and color table
Miscellaneous extensions	Polygon offset, vertex array, NURBS tesselator, and object space tesselator

Extensions Added:

ARB_depth_texture
ARB_fragment_program
ARB_imaging
ARB_multisample
ARB_multitexture
ARB_point_parameters
ARB_shadow
ARB_shadow_ambient
ARB_texture_border_clamp
ARB_texture_compression
ARB_texture_cube_map
ARB_texture_env_add
ARB_texture_env_combine
ARB_texture_env_crossbar
ARB_texture_env_dot3
ARB_texture_mirrored_repeat
ARB_transpose_matrix
ARB_vertex_blend
ARB_vertex_buffer_object
ARB_vertex_program
ARB_window_pos
ATIX_texture_env_combine3
ATIX_texture_env_route
ATIX_vertex_shader_output_point_size
ATI_draw_buffers
ATI_element_array
ATI_envmap_bumpmap
ATI_fragment_shader
ATI_map_object_buffer
ATI_separate_stencil
ATI_texture_env_combine3
ATI_texture_float
ATI_texture_mirror_once
ATI_vertex_array_object
ATI_vertex_attrib_array_object
ATI_vertex_streams
EXT_bgra
EXT_blend_func_separate
EXT_clip_volume_hint
EXT_compiled_vertex_array
EXT_copy_texture
EXT_draw_range_elements
EXT_fog_coord
EXT_multi_draw_arrays
EXT_point_parameters
EXT_polygon_offset
EXT_rescale_normal
EXT_secondary_color
EXT_separate_specular_color
EXT_stencil_wrap
EXT_subtexture
EXT_texgen_reflection

EXT_texture
EXT_texture3D
EXT_texture_compression_s3tc
EXT_texture_cube_map
EXT_texture_edge_clamp
EXT_texture_env_add
EXT_texture_env_combine
EXT_texture_env_dot3
EXT_texture_filter_anisotropic
EXT_texture_lod_bias
EXT_texture_object
EXT_texture_rectangle
EXT_vertex_array
EXT_vertex_shader
HP_occlusion_test
INGR_interlace_read
NV_blend_square
NV_occlusion_query
NV_point_sprite
NV_texgen_reflection
S3_s3tc
SGIS_generate_mipmap
SGIS_multitexture
SGIS_pixel_texture
SGIS_texture_color_mask
SGIS_texture_lod
SGIX_async
SGIX_async_pixel
SGIX_blend_alpha_minmax
SGIX_convolution_accuracy
SGIX_fragment_lighting
SGIX_resample
SGIX_scalebias_hint
SGIX_subsample
SGIX_texture_coordinate_clamp
SGIX_vertex_preclip
SUN_multi_draw_arrays

Record of Revision

Version	Description
001	1996 Original publication.
002	1998 Updated to support OpenGL 1.1.
003	March 2005 Updated to support OpenGL 1.3 and extensions to support Onyx4 and Silicon Graphics Prism systems.

Contents

Figures

Tables

Examples

About This Guide

OpenGL on Silicon Graphics Systems explains how to use the OpenGL graphics library on Silicon Graphics systems. This guide expands the description of OpenGL programming presented in the book *OpenGL Programming Guide*, which describes aspects of OpenGL that are implementation-independent.

This guide describes the following major topics:

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

Silicon Graphics Visualization Systems

Though some items in this guide apply to all Silicon Graphics visualization systems, this guide explicitly addresses the following families of visualization systems:

- Silicon Graphics VPro systems (Fuel and Tezro systems)
- Silicon Graphics InfinitePerformance systems
- Silicon Graphics InfiniteReality systems
- Silicon Graphics Onyx4 UltimateVision systems
- Silicon Graphics Prism systems (Linux systems)

What This Guide Contains

This guide consists of the following chapters and 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” describes extensions that facilitate management of buffers and similar resources. Most of these extensions are GLX extensions.
- Chapter 7, “Vertex Processing Extensions” explains how to use vertex processing extensions.
- Chapter 8, “Texturing Extensions”explains how to use the texturing extensions, providing example code as appropriate.
- Chapter 9, “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 10, “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 11, “Video Extensions” discusses extensions that can be used to enhance OpenGL video capabilities.

- Chapter 12, “Miscellaneous OpenGL Extensions” explains how to use the instruments and list priority extensions as well as two extensions to GLU.
- Chapter 13, “Vertex and Fragment Program Extensions” explains how to use the programmable shading extensions introduced in Onyx4 and Silicon Graphics Prism graphics systems.
- Chapter 14, “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 15, “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 16, “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 17, “Tuning Graphics Applications: Examples” provides a detailed discussion of the tuning process for a small example program. It also provides a code fragment that is helpful for drawing pixels fast.
- Chapter 18, “System-Specific Tuning” provides information on tuning some specific Silicon Graphics systems: InfiniteReality, Onyx4, and Silicon Graphics Prism systems.
- Appendix A, “Benchmarks” lists a sample benchmarking program.
- Appendix B, “Benchmarking Libraries: libpdb and libisfast” discusses two libraries you can use for benchmarking drawing operations and maintaining a database of the results.
- Appendix C, “System Support for OpenGL Versions and Extensions” list the OpenGL core versions and all extensions currently supported on VPro, InfinitePerformance, InfiniteReality, Onyx4, and Silicon Graphics Prism systems.
- Appendix D, “XFree86 Configuration Specifics” provides information about customizing the XF86Config file for Silicon Graphics Prism systems.

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" are not 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 are a newcomer to any of these topics, see the references listed in section "Background Reading" on page xl.

Background Reading

The following books provide background and complementary information for this guide. Bibliographical information or the SGI document number is provided. Books available online from SGI are marked with **(S)**. For access information, see section "Obtaining Publications" on page xlii.

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.

  Note that while still useful, this book does not describe the newer features of GLX 1.3.

- Dave Shreiner, OpenGL Architecture Review Board, Mason Woo, Jackie Neider and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4*. Reading, MA: Addison Wesley Longman Inc., 2003. ISBN 0-321-17348-1.

- Dave Shreiner, OpenGL Architecture Review Board. *OpenGL 1.4 Reference Manual (4th Edition). The Official Reference Document for OpenGL, Version 1.4*. Reading, MA: Addison Wesley Longman Inc., 2004. ISBN 0-321-17383-X.

- *OpenGL Porting Guide* (007-1797-030) **(S)**

- *Silicon Graphics Onyx4 UltimateVision User's Guide (007-4634-xxx)* **(S)**

- *Silicon Graphics UltimateVision Graphics Porting Guide* (007-4297-001) **(S)**

- *Silicon Graphics Prism Visualization System User's Guide (007-4701-xxx)* **(S)**

- *Obtaining Maximum Performance on Silicon Graphics Prism Visualization Systems* (007-4271-*xxx*) **(S)**

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, 1992. **(S)**
  - *Volume Two. Xlib Reference Manual*. Sebastopol, CA: O'Reilly & Associates, 1992.
  - Nye, Adrian, and Tim O'Reilly. *Volume Four. X Toolkit Intrinsics Programming Manual*. Sebastopol, CA: O'Reilly & Associates, 1992. **(S)**
  - Flanagan, David (ed). *Volume Five. X Toolkit Intrinsics Reference Manual*. Sebastopol, CA: O'Reilly & Associates, 1992.
  - 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. **(S)**
- Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.2*. Englewood Cliffs, NJ: Prentice Hall, 1993. **(S)**
- 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. **(S)**

Other Sources

- Kane, Gerry. *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice Hall. 1989.
- *MIPS Compiling and Performance Tuning Guide*. 007-2479-001. **(S)**

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- On IRIX, you can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man` <*title*> on a command line.
- SGI ProPack for Linux documentation and all other documentation included in the RPMs on the distribution CDs can be found on the CD titled *SGI ProPack 3 for Linux - Documentation CD*. To access the information on the documentation CD, open the `index.html` file with a web browser. After installation, all SGI ProPack for Linux documentation (including `README.SGI`) is in the directory `/usr/share/doc/sgi-propack-3.0`.

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:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
function	This bold font indicates a function or method name. Parentheses are also appended to the name.

Convention	Meaning
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
GUI element	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists.

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*.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library webpage:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, CA 94043-1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

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

The OpenGL graphics library is not limited to a particular window system. 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, Microsoft Windows, Mac OS X, and other major window systems. If you want 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 IRIX Interactive Desktop environment is based on X, as is the Silicon Graphics widget set, IRIS IM. IRIS IM is the Silicon Graphics port of OSF/Motif.

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

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" on page 11. More detailed information is in Appendix D, "OpenGL Extensions to the X Window System" of the *OpenGL Programming Guide.* The `glxintro` man 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 describes 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 then describes the Xlib library, which is more primitive but offers more control. Most application developers usually write at a higher level than Xlib, but you may find it helpful to understand the basic facts about the lower levels 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 1-1 illustrates the layering:

- IRIS ViewKit (only supported on IRIX systems) 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.

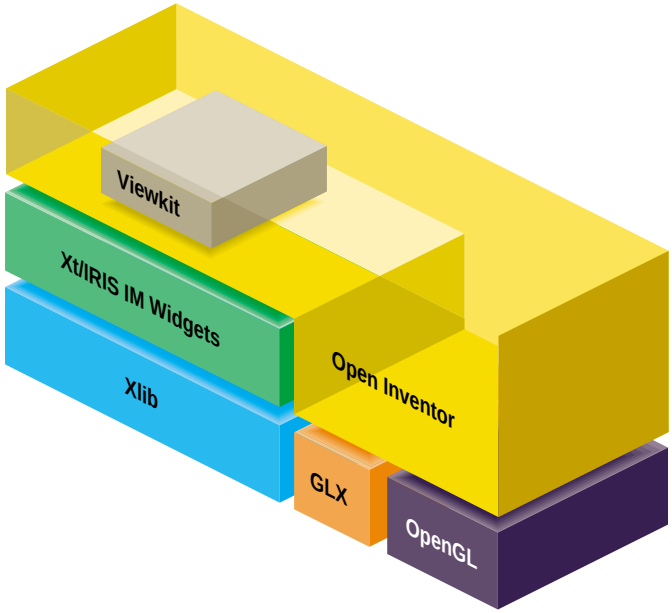


Figure 1-1 How X, OpenGL, and Toolkits Are Layered

Note: If you write an application on IRIX using IRIS Viewkit or Open Inventor, the graphical user interface will be visually consistent with the IRIX Interactive Desktop.

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" on page xl 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.

Porting Applications between IRIX and Linux

Not all of the toolkits just described are available on all Silicon Graphics platforms and if you are targeting both IRIX and Linux, you should be aware of the differences. IRIS ViewKit is only supported on IRIX systems, but Integrated Computer Solutions Incorporated (ICS) makes a commercial version of ViewKit for Linux and other platforms. The IRIS IM widget set includes widgets specific to SGI and supported only on IRIX. However, the OSF/Motif implementation on Linux supports most of the same functionality.

In addition to the toolkits and widget sets described earlier, similar GUI functionality is available from open source packages such as the Gnome Toolkit (GTK), Qt from Trolltech, and many others. SGI provides industry-standard versions of some of these packages with SGI Linux systems, and some are also available prebuilt for IRIX through the IRIX Freeware site, http://freeware.sgi.com/. Although SGI does not recommend any specific alternative, you may find these toolkits useful.

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. For example, the suffix ARB is used for extensions reviewed and approved by the OpenGL Architecture Review Board. ARB extensions are likely to be more widely supported on different vendor platforms than are any other type of extension, as they represent a consensus of the graphics industry. For a complete listing of suffixes, see Chapter 5, "Introduction to OpenGL Extensions".

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

- Chapter 5, “Introduction to OpenGL Extensions”
- Chapter 6, “Resource Control Extensions”
- Chapter 7, “Vertex Processing Extensions”
- Chapter 8, “Texturing Extensions”
- Chapter 9, “Rendering Extensions”
- Chapter 10, “Imaging Extensions”
- Chapter 11, “Video Extensions”
- Chapter 12, “Miscellaneous OpenGL Extensions”
- Chapter 13, “Vertex and Fragment Program 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 rarely be suitably tuned.

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

- Chapter 15, “Tuning Graphics Applications: Fundamentals”
- Chapter 16, “Tuning the Pipeline”
- Chapter 17, “Tuning Graphics Applications: Examples”
- Chapter 18, “System-Specific Tuning”

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* or gdb.
- The CASE tools are only available on IRIX for general-purpose debugging. For more information on the CASE tools, see *ProDev WorkShop and MegaDev Overview* and *CASEVision/Workshop User's Guide.*

Maximizing Performance With OpenGL Performer

The OpenGL 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 OpenGL 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.

For OpenGL Performer documentation, see the SGI Technical Publications Library, http://docs.sgi.com.

Location of Example Source Code (IRIX-Specific)

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.

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. Following the background information is a simple example program that displays OpenGL code in an X window. This chapter uses the following topics:

Background and Terminology

To effectively integrate your OpenGL program with the X Window System, you need to understand the basic concepts described in the following 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" on page xl.

X Window System on Silicon Graphics Systems

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

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 Servers

There are two different X servers provided depending on the operating system and type of graphics supported:

- Xsgi

  For traditional IRIX graphics systems such as VPro, InfinitePerformance, and InfiniteReality, Silicon Graphics uses its own X server, called Xsgi.

- XFree86

  For IRIX Oynx4 systems and all Linux systems, Silicon Graphics uses an X server from the open source XFree86 project. This server contains newer X extensions such as RENDER but does not support all of the extensions of the Xsgi server.

While both Xsgi and XFree86 are based on the X Consortium X11R6 source code base, Xsgi 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 extensions, and simultaneous displays on multiple graphics monitors. Specifically for working with OpenGL programs, Silicon Graphics offers the GLX extension described 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 listed 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. The Xsgi and XFree86 servers shipped with Silicon Graphics systems all support GLX. GLX is both an API and an X extension protocol for supporting OpenGL. GLX routines provide basic interaction between X and OpenGL. Use them, for example, to create a rendering context and bind it to a window.

Compiling With the GLX Extension

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

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

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

Library	Header	Link Line
OpenGL	GL/gl.h	-lGL
GLU	GL/glu.h	-lGLU
GLX	GL/glx.h	-lGL (includes GLX and OpenGL)
X11	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 describes the following concepts you will encounter throughout this guide:

- "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 4-1 on page 71 for information about the visuals and their supported OpenGL rendering modes. Framebuffer configurations, or FBConfigs, allow additional combinations. For details, see the section "Using Visuals and Framebuffer Configurations" on page 71.

GLX overloads X visuals to include both the standard X definition of a visual and information specific to OpenGL 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, do the following:

- Use *`xdpyinfo`* to display all the X visuals your system supports.
- Use *`glxinfo`* or *`findvis`* to find visuals that can be used with OpenGL.

  The *`findvis`* command (only available on SGI IRIX systems) can actually look for available visuals with certain attributes. See the `xdpyinfo`, `glxinfo`, and `findvis` man 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 OpenGL 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 and Framebuffer Configurations" on page 71. Table 4-1 on page 71 illustrates 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 into which X can draw, either a window or a pixmap. An exception is a pixel buffer (pbuffer), which is a GLX drawable but cannot be used for X rendering. A GLX drawable is something into which both X and OpenGL can draw, either an OpenGL capable window or a GLX pixmap. (A GLX pixmap is a handle to an X pixmap that is allocated in a special way; see Figure 4-2 on page 96.) Different ways of creating a GLX drawable are discussed in "Drawing-Area Widget Setup and Creation" on page 31, "Creating a Colormap and a Window" on page 45, and "Using Pixmaps" on page 96.

Pbuffers were promoted from the SGIX_pbuffer extension to GLX 1.1 into a standard part of GLX 1.3, which is supported on all current Silicon Graphics visualization systems. So, the SGIX_pbuffer extension is no longer described in detail in this document.

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.

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

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

Resources 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 or a different process through the use of FBConfigs. For details, see the section "Using Visuals and Framebuffer Configurations" on page 71.

Note: The term *resource* can, in other X-related contexts, refer to items handled by the Resource Manager. They are items that users can customize for their own use. These resources are user data in contrast to the server data described in this section.

X Window Colormaps

A colormap maps pixel values from the framebuffer to intensities on the 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 so 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 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" on page 83. Colormaps are also discussed in detail in O'Reilly, Volume One.

Libraries, Toolkits, and Tools

This section first describes 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.

If you develop on IRIX, 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 IRIX Interactive 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. If you develop for cross-platform environments or only for Linux environments, use those features of OSF/Motif that are not specific to SGI or use other toolkits such as GTK or Qt.

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 does not come 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” on page 5.

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" on page xl for full bibliographic information and for pointers to additional documents about Motif and IRIS IM. The 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. For more information, see "Open Inventor" on page 3, "IRIS ViewKit" on page 4, and "Porting Applications between IRIX and Linux" on page 5.

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).

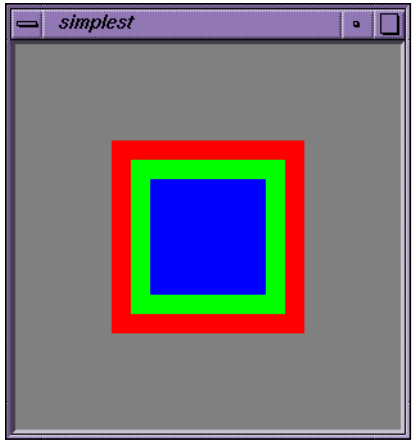


Figure 2-1 Display From `simplest.c` Example Program

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 described in a separate section:

- "Opening the X Display"
- "Selecting a Visual"
- "Creating a Rendering Context"
- "Creating the Window" (described with program examples in "Drawing-Area Widget Setup and Creation" on page 31 and "Creating a Colormap and a Window" on page 45)
- "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 described in "Input Handling With Widgets and Xt" on page 37 and, for Xlib programming, "Xlib Event Handling" on page 48.

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 2-1 above. **XtOpenApplication()** actually opens the display and performs some additional setup:

- Initializing Xt
- Opening an X server connection
- Creating an X context (not a GLX context) for the application
- Creating an application shell widget
- Processing command-line options
- Registering fallback resources

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

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

In most cases, NULL is appropriate for both *errorBase* and *eventBase*. See the `glXQueryExtension` man 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 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*/
     }
 }
```

GLX 1.3 is supported on all current Silicon Graphics platforms under IRIX 6.5 and Linux. In addition to providing a few new functions and a mechanism for using extensions (introduced in GLX 1.1), GLX 1.3 promoted the SGIX_fbconfig, SGIX_pbuffer, and SGIX_make_current_read GLX extensions to become standard parts of the core 1.3 API.

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. The following is the function's format:

```
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 can specify the following:
  - Whether to use RGBA or color-index mode (depending on whether `GLX_RGBA` is `True` or `False`)
  - Whether to use double-buffering or not (depending on the value of `GLX_DOUBLEBUFFER`)
  - How deep the depth buffer should be (depending on the value of `GLX_DEPTH_SIZE`)

  In Example 2-1 on page 17, 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` man 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 from 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 one 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.

GLX 1.3 allows you to create and choose a glXFBConfig construct, which packages GLX drawable information, for use instead of a visual.

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. Therefore, before you can draw anything, you must create a rendering context for OpenGL using **glXCreateContext()**, which has the following function format:

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

The following describes the arguments:

dpy	The display you have already opened.
vis	The visual you have chosen with **glXChooseVisual()**.
sharedList	A context for sharing display lists or `NULL` to not share display lists.
direct	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" on page 98.

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 described, with program examples, in "Drawing-Area Widget Setup and Creation" on page 31 and "Creating a Colormap and a Window" on page 45.

Binding the Context to the Window

If you are working with Xlib, bind the context to the window by calling **glXMakeCurrent().** Example 3-2 on page 43 is 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: Do not make OpenGL calls until the context and window have been bound (made current).

For each thread of execution, only one context can be bound to a single window or pixmap.

Note: GLX 1.3 allows you to attach separate read and write drawables to a GLX context. For details, see section "SGI_make_current_read—The Make Current Read Extension" on page 114.

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 have 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 does not 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 to 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` man pages.

Integrating OpenGL Programs With X—Summary

Table 2-2 summarizes the steps that are needed to integrate an OpenGL program with the X Window System. While some functions differ in IRIS IM and Xlib, note that the GLX functions are usually common.

Table 2-2 Integrating OpenGL and X

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

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

Compiling With OpenGL and Related Libraries

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

Link Lines for Individual Libraries

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

`-lGL`	OpenGL and GLX routines.
`-lX11`	Xlib, X client library for X11 protocol generation.
`-lXext`	The 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.
`-lGLw`	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. The `image` library is only supported under IRIX. Open source alternatives like `libjpeg` and `libpnm` provide image I/O functions and are better alternatives when writing code that must also run on Linux and other platforms.
`-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	`-Xmu`
With Motif and OpenGL widget	`-lGLw -lXm -lXt`

OpenGL and X: Examples

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

- "Using Widgets" on page 29 illustrates how to create a window using IRIS IM drawing-area widgets and how to handle input and other events using callbacks.
- "Using Xlib" on page 42 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 describes fonts: "Using Fonts and Strings" on page 51 describes 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 described in "Integrating Your OpenGL Program With IRIS IM" on page 16 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 does not manage. The section desribes 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 does the following:

- Provides an environment for OpenGL rendering, including a visual and a colormap.
- Provides a set of callback routines for redrawing, resizing, input, and initialization (see "Using Drawing-Area Widget Callbacks" on page 34).

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 the following 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 entries 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. Therefore, it has various defaults such as background and foreground colors. GLwMDrawingArea is **not** derived from the standard Motif drawing-area widget class. For more information, see O'Reilly Volume One or the man pages for `Core` and for `XmPrimitive`.

  Note that the default background colors provided by the widget are used during X rendering, not during OpenGL rendering. Therefore, 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 better with the program.

- GLwMDrawingArea has the creation function **GLwCreateMDrawingArea()** in the style of IRIS IM. You can also create the widget directly through Xt.

For information specific to GLwDrawingArea, see the manpage.

Drawing-Area Widget Setup and Creation

Most of the steps for writing a program that uses a GLwMDrawingArea widget are already described in "Integrating Your OpenGL Program With IRIS IM" on page 16. This section explains how to initialize IRIS IM and how to create the drawing-area widget using code fragments from the *motif/simplest.c* example program (Example 2-1 on page 17). This section has the following topics:

- "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 X server data structures for windows, pixmaps, and context described 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 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. Therefore, creating your drawing surface consists of the following two steps:

1. Create parent widgets, namely the top-level widget and a container widget. The program *motif/simplest.c,* Example 2-1 on page 17, uses a Form container widget and a Frame widget to draw the 3D box:

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

   For more information, see the man 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);
     ```

   - 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. The following is an example from *motif/simplest.c:*

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

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

Note that unlike most other Motif 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

When calling the widget creation function,there are three ways of configuring the GLwMDrawingArea widget (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.

Appropriate error handling is critical to a robust program. 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.

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, do the following:

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" on page 24).

The following callbacks are available:

Callback	Description
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. The callback reason is `GLwCR_GINIT`. Use of this callback is optional. 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 like items.

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. The callback reason is `GLwCR_INPUT`.

The input callback is a programming convenience; it provides a convenient way to catch all input events. You can often create a more modular program, however, by providing specific actions and translations in the application rather than using a single catchall callback. See "Input Handling With Widgets and Xt" on page 37 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

Using the following topics, this section explains how to perform input handling with widgets and Xt:

- "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 does not have to do it. Finally, translations allow the user to customize the application's key bindings. See "Using Actions and Translations" on page 39.

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()
```

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

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

Example 3-1 Motif Program That Handles Mouse Events

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

   switch(event->type) {
   case KeyRelease:
      XLookupString(&event->xkey, buffer, 30, &keysym, NULL);
      switch(keysym) {
      case XK_Escape:
         exit(EXIT_SUCCESS);
         break;
      default: break;
      }
      break;
    case ButtonPress:
        if (event->xbutton.button == Button2) {
            mstate |= 2;
            mx = event->xbutton.x;
            my = event->xbutton.y;
        } else if (event->xbutton.button == Button1) {
            mstate |= 1;
            mx = event->xbutton.x;
            my = event->xbutton.y;
        }
        break;
```

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

Using Actions and Translations

Actions and translations provide a mechanism for binding a key or mouse event to a function call. For example, you can structure your program to take the following actions:

- 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 specifies 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 apparent 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 on the SGI Technical Publications Library.

Creating Colormaps

By default, a widget creates a colormap automatically. For many programs, this is sufficient. However, it is occasionally necessary to create a colormap explicitly, especially when using color index mode. See "Creating a Colormap and a Window" on page 45 and "Using Colormaps" on page 83 for more information.

Widget Troubleshooting

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

Note: Additional debugging information is provided in "General Tips for Debugging Graphics Programs" on page 404.

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. The following are two solutions to this problem:

- The easiest solution is to set the following resource for the application:

  ```
  keyboardFocusPolicy: POINTER
  ```

  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 do the following:

  1. Set the following resource:

     **widget*`.traversalOn: TRUE`

     The field *widget* is the name of the widget.

  2. Whenever mouse button 1 is pressed in the widget, call the following function:

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

     Turning process traversal on causes the window to respond to traversal (it normally does not), and calling **XmProcessTraversal()** actually traverses into the widget when appropriate.

Inheritance Issues

In Xt, shell widgets include top-level windows, popup windows, and menus. Shell widgets inherit their colormap and pixel depth from their parent widget and inherit their visual from the parent window. If the visual does not match the colormap and depth, this leads to a `BadMatch X protocol` error.

In a typical IRIS IM program, everything runs in the default visual, and the inheritance from two different places does not cause problems. However, when a program uses both OpenGL and IRIS IM, it requires multiple visuals, and you must be careful. Whenever you create a shell widget as a child of a widget in a non-default visual, specify pixel depth, colormap, and a visual for that widget explicitly. This happens with menus or popup windows that are children of OpenGL widgets. See “Using Popup Menus With the GLwMDrawingArea Widget” on page 69.

If you do get a bad match 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 do not 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 does not manage. This section has the following topics:

Simple Xlib Example Program

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 that you have both available as needed. Example 3-2 lists the complete *Xlib/simplest.c* example program.

Example 3-2 Simple Xlib Example Program

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

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

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

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

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

    do {
        char buf[31];
        KeySym keysym;

        XNextEvent(dpy, &event);
```

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

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

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

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

        /* get an appropriate visual */
        vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributeList);
```

```
    if (!vi) error(argv[0], “no suitable visual”);

    /* create a GLX context */
    cx = glXCreateContext(dpy, vi, 0, GL_TRUE);
    /* create a colormap */
    swa.colormap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                                   vi->visual, AllocNone);
    /* create a window */
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | StructureNotifyMask | KeyPressMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 300,
                        300, 0, vi->depth, InputOutput, vi->visual,
                        CWBorderPixel|CWColormap|CWEventMask, &swa);
    XStoreName(dpy, win, “simplest”);
    XMapWindow(dpy, win);

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

    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 3-2 calls **XCreateColormap()** in the following manner:

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

The parameters specify the display, window, visual, and the number of colormap entries to allocate. The *alloc* parameter can have the special value `AllocAll` or `AllocNone`.

While it is easy to simply call **XCreateColormap()**, you are encouraged to share colormaps. See Example 4-2 on page 85 for 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" on page 83. For information on overlays, which use a visual with a transparent pixel, see "Using Overlays" on page 62.

After creating a colormap, you can 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 the symbolic constants that specify the attributes you have set. Here's how Example 3-2 does it in the following way:

```
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. However, the following three are noteworthy:

class — Indicates whether the window is `InputOnly` or `InputOutput`.

Note: `InputOnly` windows cannot be used with GLX contexts.

valuemask — Specifies which window attributes are provided by the call.

attributes — Specifies the settings for the window attributes. The XSetWindowAttributes structure contains a field for each of the allowable attributes.

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

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

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

XSetWMProperties() provides a convenient interface for setting a variety of important window properties at once. It merely calls a series of other property-setting functions, passing along the values you pass in. For more information, see the man 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()** to pass the display, the top-level window, a list of windows whose colormaps should be installed, and the number of windows in the list.

The list of windows should include one window for each colormap, including the top-level window's colormap (normally represented by the top-level window). For a

typical OpenGL program that does not use overlays, the list contains two windows: the OpenGL window and the top-level window. The top-level window should normally be last in the list. Xt programs may use **XtSetWMColormapWindows()** instead of **XSetWMColormapWindows()**, which uses widgets instead of windows.

Note: The program must call **XSetWMColormapWindows()** even if it is using a TrueColor visual. Some hardware simulates TrueColor through the use of a colormap. Even though the application does not interact with the colormap directly, it is still there. If you do not 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 describes 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 on the SGI Technical Publications Library.

Handling Mouse Events

To handle mouse events, your program first has to request them and 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 3-3 shows how the mouse processing, along with the other event processing, is defined.

Example 3-3 Event Handling With Xlib

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

    do {
        char buf[31];
        KeySym keysym;
        XNextEvent(dpy, &event);
        switch(event.type) {
        case Expose:
            redraw = 1;
            break;
        case ConfigureNotify:
            glViewport(0, 0, event.xconfigure.width,
                       event.xconfigure.height);
            redraw = 1;
            break;
        case KeyPress:
            (void) XLookupString(&event.xkey, buf, sizeof(buf),
                   &keysym, NULL);
            switch (keysym) {
            case XK_Escape:
                exit(EXIT_SUCCESS);
            default:
                break;
            }
        case ButtonPress:
            if (event.xbutton.button == Button2) {
                mstate |= 2;
                mx = event.xbutton.x;
                my = event.xbutton.y;
            } else if (event.xbutton.button == Button1) {
                mstate |= 1;
                mx = event.xbutton.x;
                my = event.xbutton.y;
            }
            break;
        case ButtonRelease:
            if (event.xbutton.button == Button2)
                mstate &= ~2;
            else if (event.xbutton.button == Button1)
```

```
                mstate &= ~1;
            break;
        case MotionNotify:
            if (mstate) {
                omx = mx;
                omy = my;
                mx = event.xbutton.x;
                my = event.xbutton.y;
                update_view(mstate, omx,mx,omy,my);
                redraw = 1;
            }
            break;
        default:
            break;
        }
    } while (XPending(dpy));
    return redraw;
}
```

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

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

Exposing a Window

When a user selects a window that has been completely or partly covered, the X server generates one or more Expose events. It is difficult to determine exactly what was drawn in the now-exposed region and redraw only that portion of the window. Instead, OpenGL programs usually just redraw the entire window.

If redrawing is not an acceptable solution, the OpenGL program can do all your rendering into a GLXPixmap instead of directly to the window; then, any time the program needs to redraw the window, you can simply copy the GLXPixmap's contents into the window using **XCopyArea()**. For more information, see "Using Pixmaps" on page 96.

Note: Rendering to a GLXPixmap is much slower than rendering to a window and may not allow access to many features of the graphics hardware.

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

Using Fonts and Strings

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

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

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

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

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

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

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

Note: You can also use the `glc` library, which sits atop of OpenGL, for fonts and strings. The library is not specific to GLX and provides other functions in addition to **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 ();
}
```

OpenGL and X: Advanced Topics

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

Using Animations

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

Providing example code as appropriate, this section uses the following topics to describe double-buffered animation inside an X Window System environment:

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

- "Controlling an Animation With Timeouts"

Xt provides two mechanisms that are suited for continuous animation:

- The section "Controlling an Animation With Workprocs" on page 57 describes the fastest animation possible. If you use workprocs, the program swaps buffers as fast as possible; this 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 do not 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.
- The section "Controlling an Animation With Timeouts" on page 60 describes constant-speed animation. Animations that use timeouts compete on even footing with other Xt events; the animation will not 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; therefore, 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 do not 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 the front buffer and 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 the buffer swap is instantaneous during the vertical retrace of the monitor. As a result, there are no

tearing artifacts; that is, you do not 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 are 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.

Note that users like uniform frame rates such as 60 Hz, 30 Hz, or 20 Hz. Animation may otherwise look jerky. A slower consistent rate is therefore preferable to a faster but inconsistent rate. For additional information about optimizing frame rates, see "Optimizing Frame Rate Performance" on page 419. See "SGI_swap_control—The Swap Control Extension" on page 287 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 executes quickly. 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 is called again when there is idle time.

To explicitly remove a workproc, call **XtRemoveWorkProc()**. The following shows the syntax for the add and remove functions:

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

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

Workproc Example

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

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

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

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

```
static void
map_change(Widget w, XtPointer clientData, XEvent *event, Boolean
                                                              *cont) {
    switch (event->type) {
    case MapNotify:
    /* resume animation if we become mapped in the animated state */
        if (state.animate_wpid != 0)
             state.animate_wpid = XtAppAddWorkProc(state.appctx,
                                      redraw_proc, &state.glxwidget);
        break;
    case UnmapNotify:
    /* don't animate if we aren't mapped */
        if (state.animate_wpid) XtRemoveWorkProc(state.animate_wpid);
        break;
    }
}
```

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

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

The **redraw_proc()** function, in turn, calls **draw_scene()**, which swaps the buffers. Note that this program does not 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.

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

  From 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.

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

  From time_animate:
  ```
  static struct {        /* global UI variables; keep them together */
      XtAppContext appctx;
  ```

```
    Widget glxwidget;
    Boolean direct;
    XtIntervalId animate_toid;
} state;
```

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

  From work_animate:

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

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

  From time_animate:

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

    switch (entry) {
    case 0:
        if (state.animate_toid) {
            XtRemoveTimeOut(state.animate_toid);
            state.animate_toid = 0;
        } else {
            /* register timeout */
```

```
            state.animate_toid = XtAppAddTimeOut(state.appctx,
                                TIMEOUT, redraw_proc, &state.glxwidget);
        }
        break;
    case 1:
        exit(EXIT_SUCCESS);
        break;
    default:
        break;
    }
}
```

- The **redraw_proc()** function has to register a new timeout each time it 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.glxwidget);
}
```

Using Overlays

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

- “Introduction to Overlays”
- “Creating Overlays”
- “Overlay Troubleshooting”
- “Rubber Banding”

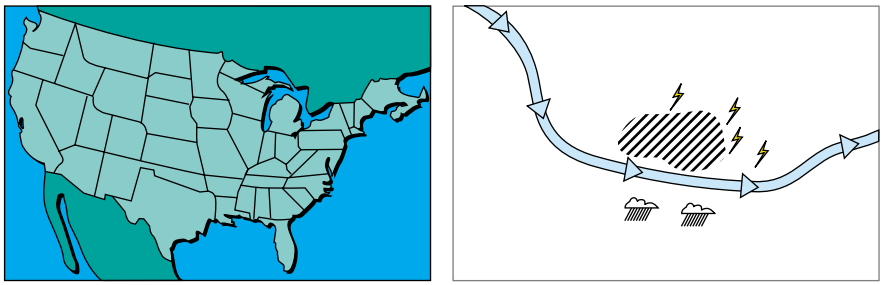
Introduction to Overlays

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.

Notes:

- Transparency discussed here is distinct from transparency effects based on alpha buffer blending. See the section "Blending" in Chapter 7, "Blending, Anti-Aliasing, and Fog," in the *OpenGL Programming Guide.*
- On Silicon Graphics systems running the XFree86 server (for example, Onyx4 and Silicon Graphics Prism systems), you must configure the XFree86 server to support overlay planes. Refer to the platform-specific documentation for the details of configuring XFree86.



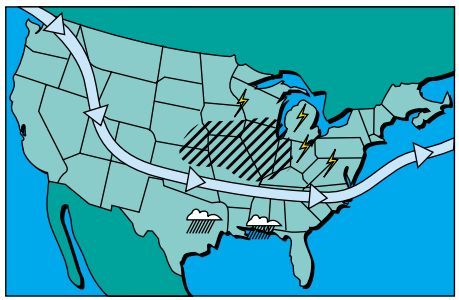


Figure 4-1 Overlay Plane Used for Transient Information

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 to allow 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 the following 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" on page 65 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.

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" on page 117 for more information.

Note: This example uses the visual info extension, which is supported on all current Silicon Graphics graphics systems. The visual info extension has also been promoted to a core feature of GLX 1.3. With new applications, use the GLX 1.3 interface instead of the extension.

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`.

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

```
                    None };
```

2. Create a frame and form, create the window widget, and attach 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. Using the overlay visual attributes specified in step 1 and attaching it to the same form as the window, create the overlay widget. This assures that when the window is moved or resized, the overlay is moved or resized 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. Therefore, you should call **XSetWMColormapWindows()** to create the colormap, populate it with colors, and to install it.

  Note: Overlays on Silicon Graphics systems reserve pixel 0 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 0.

- **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 can 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**

  Most Silicon Graphics systems support 8-bit overlay planes. In some cases, as with Onyx4 and Silicon Graphics Prism systems, overlay planes and stereo visuals may be mutually exclusive, as chosen when the X server is initialized.

- **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 events on the overlay window as well.

- **Missing overlay visuals**

  On Silicon Graphics systems running the XFree86 server (for example, Onyx4 and Silicon Graphics Prism systems), there may be no overlay planes configured. Hence, there will be no visuals at framebuffer levels other than 0. If **glXChooseVisual()** returns no visuals when `GLX_LEVEL` is specified as `1` in the attribute list, the application must use a different strategy to display content that would otherwise go in the overlay planes.

- **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 must manage it yourself.

The following procedure 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.

   This action causes Expose events to be sent to other windows in the overlay plane.

Using Popup Menus With the GLwMDrawingArea Widget

Popups 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); this leads to a `BadMatch X protocol` error. For more details and for information on finding the error, see "Inheritance Issues" on page 41.

The following are two ways to work around this problem:

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

Example 4-1 Popup Code Fragment

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

    XmString button_labels[XtNumber(button_types)];
```

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

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

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

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

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

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

    create_popup(frame);
```

```
    XtRealizeWidget(toplevel);

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

    XtAppMainLoop(app);
}
```

Using Visuals and Framebuffer Configurations

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

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

Some Background on Visuals

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

Table 4-1 lists the X visuals supported for different types of OpenGL rendering, indentifies the Silicon Graphics systems supporting the X visuals, and indicates whether the colormaps for those visuals are writable or not.

Table 4-1 X Visuals and Supported OpenGL Rendering Modes

OpenGL Rendering Mode	X Visual	Writable Colormap?	Supporting Systems
RGBA	TrueColor	No	All
RGBA	DirectColor	Yes	Onyx4 and Silicon Graphics Prism systems
color index	PseudoColor	Yes	All except Onyx4 and Silicon Graphics Prism systems

Table 4-1 X Visuals and Supported OpenGL Rendering Modes **(continued)**

OpenGL Rendering Mode	X Visual	Writable Colormap?	Supporting Systems
color index	StaticColor	No	Not supported
Not supported	GrayScale	Yes	Not supported
Not supported	StaticGray	No	Not supported

Depending on the available hardware and software, an X server can provide multiple visuals. Each server has a default visual, which 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 where your program will run , it is best to find out what visual classes are available on a case-by-case basis.

- From the command line, use the *xdpyinfo* command for a list of all visuals the server supports.
- Usethe *glxinfo* or *findvis* command to find visuals that are capable of OpenGL rendering. The *findvis* command can actually look for available visuals with attributes you specify. See the man 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 OpenGL 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 do not have to worry about things like colormap installation more than once in the application. However, if you use the GLwMDrawingArea widget, it does colormap installation for you; see "Drawing-Area Widget Setup and Creation" on page 31.

- 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 will not 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" on page 41). You must explicitly specify the visual depth and colormap when creating a menu. For cascading menus, specify depth and colormap separately for each pane.

Using Framebuffer Configurations

The framebuffer configuration functions in GLX 1.3 are analogous to GLX visuals but provide the following additional features:

- They introduce a new way to describe the capabilities of a GLX drawable—that is, to describe the resolution of color buffer components and the type and size of ancillary buffers by providing a GLXFBConfig construct (also called an *FBConfig*).
- They relax the "similarity" requirement when associating a current context with a drawable.
- They support RGBA rendering to one- and two-component windows (luminance and luminance alpha rendering) and GLX pixmaps as well as pixel buffers (pbuffers). Section "Using Pixel Buffers" on page 90 describes pbuffers.

The following are reasons to use framebuffer configurations:

- To use pbuffers.
- To render luminance data to a TrueColor visual.

- To replace **glXChooseVisual()**, because framebuffer configurations provide visual selection for all GLX drawables, including pbuffers, and incorporates the visual info and visual rating extensions.

This section briefly describes the following features framebuffer configurations provide:

- "Describing a Drawable With a GLXFBConfig Construct (FBConfig)"
- "Less-Rigid Similarity Requirements When Matching Context and Drawable"
- "Less-Rigid Match of GLX Visual and X Visual"

Describing a Drawable With a GLXFBConfig Construct (FBConfig)

Currently, GLX overloads X visuals so that they have additional buffers and other characteristics needed for OpenGL rendering. FBConfigs package GLX drawables by defining a new construct, a GLXFBConfig, which 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 FBConfigs, if you associate a drawable with a GLX context by calling **glXMakeCurrent()**, the two have to be similar—that is, created with the same visual. FBConfigs relax the requirement; they only require 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 created the drawable. 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` or `GLX_COLOR_INDEX_TYPE`.

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. FBConfigs make 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, FBConfigs allow RGBA rendering semantics to be used with pseudo-color and gray-scale displays. A particularly useful application of FBConfigs is that they allow you to work with single-component images with texture mapping, then use a pseudo-color visual to map the luminance values to color.

FBConfig Constructs

An 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 an FBConfig Construct

Use **glXGetFBConfigs()** to get a list of all FBConfigs that are available on the specified screen. The format of the function is as follows:

```
GLXFBConfig *glXGetFBConfigs(Display *dpy, int screen,int *nitems)
```

The number of FBConfigs returned is stored in the value *nitems*.

Use **glXChooseFBConfig()** to get FBConfig constructs that match a specified list of attributes:

```
GLXFBConfig *glXChooseFBConfig(Display *dpy, int screen,
                                        const int *attrib_list, int *nitems)
```

Like **glXGetFBConfigs()**, function **glXChooseFBConfig()** returns an array of FBConfigs that are available on the specified screen if *attrib_list* is `NULL`; otherwise, this call returns

an array of FBConfigs that match the specified attributes. Table 4-2 shows only attributes specific to FBConfigs; additional attributes are listed on the `glXChooseVisual` man page.

Table 4-2 Visual Attributes Introduced by the FBConfigs

Attribute	Type	Description
`GLX_DRAWABLE_TYPE`	bitmask	Mask indicating which GLX drawables are supported. Valid bits are `GLX_WINDOW_BIT` and `GLX_PIXMAP_BIT`.
`GLX_RENDER_TYPE`	bitmask	Mask indicating which OpenGL rendering modes are supported. Valid bits are `GLX_RGBA_BIT` and `GLX_COLOR_INDEX_BIT`.
`GLX_X_RENDERABLE`	boolean	True if X can render to drawable.
`GLX_FBCONFIG_ID`	XID	XID of the 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, larger behaves exactly smaller.

exact — Only FBConfigs whose corresponding attribute value exactly matches the requested value are considered.

mask — For an FBConfig 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.

Table 4-3 illustrates how each attribute is matched. Note that "No effect" means that the default behavior is to have no preference when searching for a matching FBConfig.

Table 4-3 FBConfig Attribute Defaults and Sorting Criteria

Attribute	Default	Sorting Criteria
GLX_BUFFER_SIZE	0	Smaller
GLX_LEVEL	0	Smaller
GLX_DOUBLEBUFFER	No effect	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_ARB	0 if GLX_SAMPLES_ARB = 0; otherwise, 1.	Smaller
GLX_SAMPLES_ARB	0	Smaller
GLX_X_VISUAL_TYPE	No effect	Exact
GLX_TRANSPARENT_TYPE	GLX_NONE	Exact
GLX_TRANSPARENT_INDEX_VALUE	No effect	Exact
GLX_TRANSPARENT_RED_VALUE	No effect	Exact

Table 4-3 FBConfig Attribute Defaults and Sorting Criteria **(continued)**

Attribute	Default	Sorting Criteria
GLX_TRANSPARENT_GREEN_VALUE	No effect	Exact
GLX_TRANSPARENT_BLUE_VALUE	No effect	Exact
GLX_TRANSPARENT_ALPHA_VALUE	No effect	Exact
GLX_VISUAL_CAVEAT	GLX_NONE	Exact if specified; otherwise, minimum
GLX_DRAWABLE_TYPE	GLX_WINDOW_BIT	Mask
GLX_RENDER_TYPE	GLX_RGBA_BIT	Mask
GLX_X_RENDERABLE	No effect	Exact
GLX_FBCONFIG_ID	No effect	Exact

There are several uses for the **glXChooseFBConfig()** function:

- Retrieve an FBConfig with a given ID specified with GLX_FBCONFIG_ID.
- 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 **glXGetFBConfigAttrib()** 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 "Using Pixel Buffers" on page 90).

Below is a description of what happens when you call **glXChooseFBConfig()**:

- 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* does not 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" on page 82 describes the selection process.) Use **XFree()** to free the memory returned by **glXChooseFBConfig()**.

- If GLX_RENDER_TYPE 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 | GLX_COLOR_INDEX_BIT is specified as the mask, then **glXChooseFBConfig()** 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 is GLX_RGBA_BIT.

  The attribute GLX_DRAWABLE_TYPE as 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 | GLX_PIXMAP_BIT is specified as the mask for GLX_DRAWABLE_TYPE then **glXChooseFBConfig()** searches for FBConfigs that support both windows and GLX pixmaps. The default value for GLX_DRAWABLE_TYPE is GLX_WINDOW_BIT.

If an FBConfig supports windows it has an associated X visual. Use the GLX_X_VISUAL_TYPE 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 attribute is ignored if GLX_DRAWABLE_TYPE is specified in *attrib_list* and the mask that follows does not have GLX_WINDOW_BIT set.

GLX_X_RENDERABLE 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 the following function:

```
int glXGetFBConfigAttrib(Display *dpy, GLXFBConfig config,
                         int attribute, int *value)
```

If **glXGetFBConfigAttrib()** 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 value has the GLX_WINDOW_BIT bit set.

To retrieve the associated visual, call the following function:

```
XVisualInfo *glXGetVisualFromFBConfig(Display *dpy,
                                      GLXFBConfig 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.

To create a GLX rendering context, a GLX window, or a GLX pixmap using an FBConfig, call **glXCreateNewContext()**, **glXCreateWindow()**, or **glXCreatePixmap()**. Their formats follow:

```
GLXContext glXCreateNewContext( Display *dpy, GLXFBConfig config,
                                int render_type, GLXContext share_list,
                                Bool direct )

GLXWindow glXCreateWindow( Display *dpy,GLXFBConfig config,
                           Window win, const int *attrib_list)

GLXPixmap glXCreatePixmap( Display *dpy,GLXFBConfig config,
                           Pixmap pixmap, const int *attrib_list)
```

The window passed to **glXCreateWindow()** must be created with an X visual corresponding to *config*; that is, it should be created using the same visual returned by **glXGetVisualFromFBConfig()** for that FBConfig. Similarly, the pixmap passed to **glXCreatePixmap()** must have the same color depth as *config*. If these requirements are not met, creating the window or pixmap will fail with an `X BadMatch` error.

The *attrib_list* argument specifies optional additional attributes to use in creating windows or pixmaps. Currently no additional attributes are defined; so, this parameter must always be `NULL`.

These functions are analogous to the **glXCreateContext()** and **glXCreateGLXPixmap()** functions, but they use GLXFBConfigs instead of visuals. See the `glXCreateNewContext`, `glXCreateWindow`, and `glXCreatePixmap` man pages for detailed information.

To create a pixel buffer using an FBConfig, see "Using Pixel Buffers" on page 90.

How an FBConfig Is Selected

If more than one FBConfig matches the specification, they are prioritized as follows (Table 4-3 on page 78 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_ARB or GLX_SAMPLES_ARB 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 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 has GLX_RGBA_BIT set, the visual types are prioritized as follows: TrueColor, DirectColor, PseudoColor, StaticColor, GrayScale, and StaticGray.
- If only the GLX_COLOR_INDEX is set in GLX_RENDER_TYPE, visual types are prioritized as PseudoColor, StaticColor, GrayScale, and StaticGray.
- If GLX_VISUAL_CAVEAT 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” on page 119 for more information.

Related Functions

The FBConfig feature introduces the following functions:

- **glXGetFBConfigAttrib()**
- **glXGetFBConfigs()**
- **glXChooseFBConfig()**
- **glXCreateWindow()**
- **glXCreatePixmap()**
- **glXCreateNewContext()**
- **glXGetVisualFromFBConfig()**

Using Colormaps

This section describes the use of colormaps in some detail. Note that in many cases, you will not 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 by value. A colormap is a table of index-to-RGB mappings.

Note: Onyx4 and Silicon Graphics Prism systems do not support color index rendering; only RGBA mode is available.

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

The X Window System supports six different types of visuals, with each type using a different type of colormap (see Table 4-1 on page 71). 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 described in detail with example programs in Chapter 7, "Color," of O'Reilly Volume One.

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

Color Variation Across Colormaps

The same index in different X colormaps does not necessarily represent the same color. Ensure that you have the correct color index values for the colormap you are using.

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

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

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

Multiple Colormap Issues

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

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

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

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

Example 4-2 Retrieving the Default Colormap for a Visual

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

    /* be lazy; using DirectColor too involved for this example */
    if (vi->class != TrueColor)
        fatalError("no support for non-TrueColor visual");
    /* if no standard colormap but TrueColor, make an unshared one */
    status = XmuLookupStandardColormap(dpy, vi->screen, vi->visualid,
        vi->depth, XA_RGB_DEFAULT_MAP,
        /* replace */ False, /* retain */ True);
    if (status == 1) {
        status = XGetRGBColormaps(dpy, RootWindow(dpy, vi->screen),
                                  &standardCmaps, &numCmaps,
```

```
                            XA_RGB_DEFAULT_MAP);
        if (status == 1)
            for (i = 0; i < numCmaps; i++)
                if (standardCmaps[i].visualid == vi->visualid) {
                    cmap = standardCmaps[i].colormap;
                    XFree(standardCmaps);
                    return cmap;
                }
    }
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
        vi->visual, AllocNone);
    return cmap;
}
```

Choosing Which Colormap to Use

When choosing which colormap to use, follow these heuristics:

1. 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 and GLX 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. Onyx4 and Silicon Graphics Prism systems support only RGBA mode.

   ---

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).

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. The following is the syntax 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 can use a PseudoColor visual.

Lighting and antialiasing are difficult in color-index mode, and texturing and accumulation do not 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

The following is 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 and then cover the second eye while the computer displays the image for the first eye.

This section describes the following topics:

- “Stereo Rendering Background Information”
- “Performing Stereo Rendering”

Stereo Rendering Background Information

Stereo rendering is done only using quad-buffered stereo. Legacy, low-end Silicon Graphics systems support a different stereo interface referred to as *divided-screen* stereo, which is no longer described in this document.

Quad-buffered stereo uses a separate buffer for the left and right eye; this results in four buffers if the program is already using a front and back buffer for animation. Quad-buffered stereo is supported on all current Silicon Graphics systems .

For more information on stereo rendering, see the man pages for the following functions:

- **XSGIStereoQueryExtension()**
- **XSGIStereoQueryVersion()**
- **XSGIQueryStereoMode()**
- **XSGISetStereoMode()**
- **XSGISetStereoBuffer()**

Note: The *stereo* man page includes sample code fragments and pointers to sample code as well as general information on stereo rendering.

Performing Stereo Rendering

To perform stereo rendering, 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, “OpenGL and X: Getting Started” and Chapter 3, “OpenGL and X: Examples”.

   Create a window, create a context, make the context current, and so on.
5. Start the event loop.
6. Draw the stereo image as shown in the following code:

```
glDrawBuffer(GL_BACK_LEFT);
```

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

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

Using Pixel Buffers

In addition to rendering on windows and GLX pixmaps, you can render to a pixel buffer (GLXPbuffer or *pbuffer* for short). This section describes the GLX features that allow you render to pbuffers.

About GLXPbuffers

A GLXPbuffer or pbuffer is an additional non-visible rendering pbuffer for an OpenGL renderer. A pbuffer 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. Therefore, use preserved pbuffers 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 pbuffer are also static. They are deallocated only when the pbuffer 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 pbuffer, call **glXCreatePbuffer()**:

```
GLXPbuffer glXCreatePbuffer(Display *dpy, GLXFBConfig config,
                            int attrib_list)
```

This call creates a single pbuffer and returns its XID.

The parameter *attrib_list* specifies a list of attributes for the pbuffer. Note that the attribute list is defined in the same way as the list for **glXChooseFBConfig()**: attributes are immediately followed by the corresponding desired value and the list is terminated with `None`.

The following attributes can be specified in *attrib_list*:

`GLX_PBUFFER_WIDTH` Determines the pixel width of the rectangular pbuffer. This token must be followed by an integer specifying the desired width. If not specified, the default value is `0`.

`GLX_PBUFFER_HEIGHT`	Determines the pixel height of the rectangular pbuffer. This token must be followed by an integer specifying the desired height. If not specified, the default value is `0`.
`GLX_PRESERVED_CONTENTS`	If specified with a value of `False`, an volatile pbuffer is created, and its contents may be lost at any time. If this attribute is not specified or if it is specified with a value of `True`, the contents of the pbuffer are preserved, typically, by swapping out portions of the pbuffer 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.
`GLX_LARGEST_PBUFFER`	If specified with a value of `True`, the largest available pbuffer (not exceeding the requested size specified by the values of `GLX_PBUFFER_WIDTH` and `GLX_PBUFFER_HEIGHT`) will be created when allocation of the pbuffer would otherwise fail due to lack of graphics memory. If this attribute is not specified or is specified with a value of `False`, allocation will fail if the requested size is too large even if a smaller pbuffer could be successfully created. The **glXQueryDrawable()** function may be used to determine the actual allocated size of a pbuffer.

The resulting pbuffer 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 **glXCreatePbuffer()** fails to create a pbuffer due to insufficient resources, a `BadAlloc X protocol` error is generated and `NULL` is returned. If *config* is not a valid FBConfig, then a `GLXBadFBConfig` error is generated; if *config* does not support pbuffers, a `BadMatch X protocol` error is generated.

Rendering to a Pbuffer

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 man pages for **glXCreateNewContext()**, **glXMakeCurrent()**, and **glXMakeCurrentReadSGI()**.

If a pbuffer is created with `GLX_PRESERVED_CONTENTS` 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 pbuffer 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 pbuffer, but the values of the returned pixels are undefined.

Because the contents of a volatile pbuffer can be lost at any time with only asynchronous notification (using 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 do not 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 pbuffer call **glXDestroyPbuffer()**, whose format follows:

```
void glXDestroyPbuffer(Display *dpy, GLXPbuffer pbuf)
```

To query an attribute associated with a GLX drawable (GLXWindow, GLXPixmap, or GLXPbuffer), call **glXQueryDrawable()**, whose format follows:

```
void glXQueryDrawable(Display *dpy, GLXDrawable drawable, int attribute
                            unsigned int *value)
```

The `GLX_WIDTH`, `GLX_HEIGHT`, and `GLX_FBCONFIG_ID` attributes may be queried for all types of drawables. The query returns respectively the allocated pixel width, pixel height, and the XID of the FBConfig with respect to which the drawable was created.

The `GLX_PRESERVED_CONTENTS` and `GLX_LARGEST_PBUFFER` attributes are meaningful only for GLXPbuffer drawables and return the values specified when the pbuffer was created. The values returned when querying these attributes for GLXWindow or GLXPixmap drawables are undefined.

To find the FBConfig for a drawable, first retrieve the ID for the FBConfig using **glXQueryDrawable()** and then call **glXChooseFBConfig()** with that ID specified as the `GLX_FBCONFIG_ID` attribute value in the attribute list. For more details, see "Using Framebuffer Configurations" on page 74.

Directing the Buffer Clobber Event

An X client can ask to receive GLX events on a GLXWindow or GLXPbuffer by calling **glXSelectEvent()**:

```
void glXSelectEvent(Display *dpy, GLXDrawable drawable,
                    unsigned long event_mask)
```

Currently, you can only select the `GLX_BUFFER_CLOBBER_MASK` GLX event bit in *event_mask*. The event structure is as follows:

```
typedef struct {
  int event_type;           /* GLX_DAMAGED or GLX_SAVED */
  int draw_type;            /* GLX_WINDOW or GLX_PBUFFER */
  unsigned long serial;     /* Number of last request processed */
                            /* by server */
  Bool send_event;          /* True if event was generated by a */
                            /* SendEvent request */
  Display *display;         /* Display the event was read from */
  GLXDrawable drawable;     /* XID of Drawable */
  unsigned int buffer_mask; /* Mask indicating which buffers are */
                            /* affected */
  unsigned int aux_buffer;  /* Which aux buffer was affected */
  int x, y;
  int width, height;
  int count;                /* If nonzero, at least this many more */
                            /* events*/
} GLXPbufferClobberEvent;
```

A single X server operation can cause several buffer clobber events to be sent; for example, a single pbuffer 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 event commands (**XNextEvent()**, **XPending()**, and so on). The *event_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
GLX_FRONT_RIGHT_BUFFER_BIT
GLX_BACK_LEFT_BUFFER_BIT
GLX_BACK_RIGHT_BUFFER_BIT
GLX_AUX_BUFFERS_BIT
GLX_DEPTH_BUFFER_BIT
GLX_STENCIL_BUFFER_BIT
GLX_ACCUM_BUFFER_BIT
```

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 pbuffer, a buffer clobber event with *event_type* `GLX_SAVED` is generated whenever the contents of the pbuffer are swapped out to host memory. The event(s) describes which portions of the pbuffer were affected. Clients that receive many buffer clobber events referring to different save actions should consider freeing the pbuffer resource to prevent the system from thrashing due to insufficient resources.
- For a volatile pbuffer, a buffer clobber event with *event_type* `GLX_DAMAGED` is generated whenever a portion of the pbuffer becomes invalid. The client may wish to regenerate the invalid portions of the pbuffer.
- For a window, a clobber event with *event_type* `GLX_SAVED` is genererated whenever an ancillary buffer associated with the window is moved out of off-screen memory. The event indicates which color or ancillary buffers and which portions of those buffers were affected. Windows do not generate clobber events when clobbering each other's ancillary buffers—only standard X damage events.

Calling **glXSelectEvent()** overrides any previous event mask that was set by the client for the drawable. Note that it does not affect the event masks that other clients may have specified for a drawable, 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 pbuffer, call **glXGetSelectedEvent()**:

```
void glXSelectEvent(Display *dpy, GLXDrawable drawable,
                    unsigned long event_mask)
```

Related Functions

The GLX pbuffer feature provides the following functions:

- **glXCreatePbuffer()**
- **glXDestroyPbuffer()**
- **glXQueryDrawable()**
- **glXSelectEvent()**
- **glXGetSelectedEvent()**

Using Pixmaps

An OpenGL program can render to three kinds of drawables: windows, pbuffers, and pixmaps. A pixmap is an offscreen rendering area. On Silicon Graphics systems, pixmap rendering is not hardware-accelerated. Furthermore, pixmap rendering does not support all features and extensions of the underlying graphics hardware.

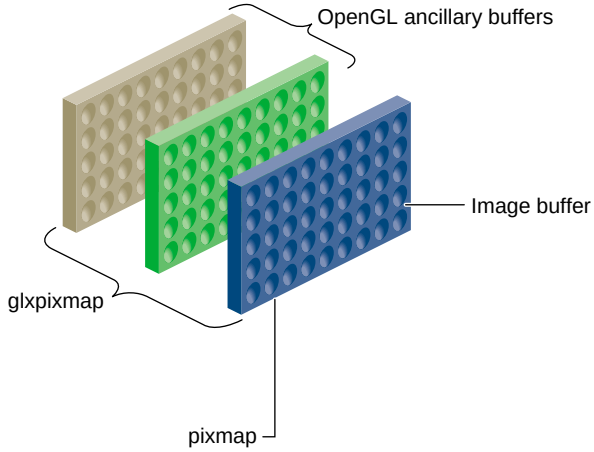


Figure 4-2 X Pixmaps and GLX Pixmaps

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 identifies 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. The following steps describe how you create and use pixmaps.

Note: Steps 1–3 and step 6 are described in detail in "Integrating Your OpenGL Program With IRIS IM" on page 16.

---

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()**, whose syntax is shown in the following:

   ```
   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 to which *vis* points (as indicated by the visual's `GLX_BUFFER_SIZE` value). Otherwise, 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

  Direct rendering contexts support rendering directly from OpenGL using 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.

- Indirect rendering

  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 (with a context switch) so that 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 X 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 penalty for mixing OpenGL and X in the same window. GLX does not constrain the order in which OpenGL commands and X requests are executed. To ensure a particular order, use the GLX commands **glXWaitGL()** and **glXWaitX()**. Use the following guidelines:

- **glXWaitGL()** prevents any subsequent X calls from executing until all pending OpenGL calls complete. When you use indirect rendering, this function does not contact the X server and is therefore more efficient than **glFinish().**
- **glXWaitX()**, when used with indirect rendering, is just the opposite: it ensures that all pending X calls complete before any further OpenGL calls are made. Also, giving this function an advantage over **XSync()** when rendering indirectly, **glXWaitX()** does not need to contact the X server.
- Remember also to batch Expose events. See "Exposing a Window" on page 50.
- 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 Microsoft Windows), certain programming practices make porting easier. The following 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 Microsoft Windows porting only, avoid naming variables with any variation of the words "near" and "far". They are reserved words in Intel *xx*86 compilers. For instance, you should avoid the names `_near`, `_far`, `__near`, `__far`, `near`, `far`, `Near`, `Far`, `NEAR`, `FAR`, and so on.
- Microsoft Windows does not have an equivalent to **glXCopyContext().**

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.

Many features initially introduced as extensions are promoted to become core features of later releases of OpenGL. When an extension is promoted in this fashion, it is documented as part of the core OpenGL 1.*x* API and usually will not be described in detail in this document.

Using the following topics, this chapter provides basic information about OpenGL extensions:

Determining Extension Availability

Function names and tokens for OpenGL extensions have a suffix describing the source of the extension—for example, **glConvolutionFilter2DEXT()** or **glColorTableSGI()**. The names of the extensions themselves (the extension strings) use prefixes—for example, SGI_color_table. The following is a detailed list of all suffixes and prefixes:

ARB	Used for extensions that have been developed and adopted by the OpenGL Architecture Review Board, the standards body controlling the OpenGL API.
EXT	Used for extensions that have been reviewed and approved by more than one OpenGL vendor.
SGI	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. Not all SGI extensions are supported on Silicon Graphics Onyx4 and Silicon Graphics Prism systems.
SGIS	Used for extensions that are found only on a subset of Silicon Graphics platforms.
SGIX	Used for extensions that are experimental: In future releases, the API for these extensions may change, or they may not be supported at all.
ATI	Used for extensions that have been developed by ATI Technologies. These extensions are found only on platforms using ATI graphics processor units (GPUs), including Silicon Graphics Onyx4 and Silicon Graphics Prism graphics systems.
ATIX	Used for experimental ATI extensions in the same fashion as SGIX.
HP, NV, etc.	Used for extensions that were initially developed by other vendors. These extensions are included for compatibility with code ported from those vendors' platforms and are not available on all Silicon Graphics platforms.

Note: When an extension is promoted to the OpenGL core, the function names and tokens have the extension suffix removed. Unless otherwise documented, the suffixed and non-suffixed forms of the functions and tokens have exactly the same meaning and use. Extensions that are promoted typically are available in both suffixed and non-suffixed forms for backwards compatibility.

How to Check for OpenGL Extension Availability

All supported extensions have a corresponding definition in *gl.h* or `glext.h` (a companion header included by `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.

Note: In this document, OpenGL extensions are listed by name without the `GL_` prefix. For example, the ABGR extension is listed under a section heading of "EXT_abgr". However, when testing for the presence of an OpenGL extension in the extensions string or in the OpenGL header files, you must use the `GL_` prefix. Note that extensions for the GLX and GLU APIs have names similarly prefixed by `GLX_` and `GL_`, and you must use these prefixes when testing for run-time or compile-time support of those extensions.

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 does not support it.
- Run-time checking ensures that the extension is supported for the OpenGL server and run-time library you are using.

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

Note that `libdl` interface allows users to dynamically load their own shared objects as needed. Applications can use this interface, particularly the **dlsym()** function, 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 graphics driver 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.

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 `define`s are found in *`glx.h`* or `glxext.h` (a companion header included by `glx.h`).
- Prefix the names of GLX extensions with `GLX_` when testing for run-time or compile-time support, just as you must prefix the names of OpenGL extensions with `GL_`.
- 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).

- All current Silicon Graphics platforms (Fuel, Tezro, InfiniteReality, InfinitePerformance, Onyx4, and Silicon Graphics Prism systems) support GLX 1.3. Most GLX extensions were promoted to the GLX 1.3 core, in some cases with minor changes in functionality. For maximum portability, applications should use the GLX 1.3 core functions and tokens instead of the extensions.

Taking these exceptions into account, adapt the process described in "How to Check for OpenGL Extension Availability" on page 103.

ARB_get_proc_address—The Dynamic Query-Function-Pointer Extension

On SGI IRIX systems, all functions defined by OpenGL and GLX extensions are exported statically from the OpenGL link library so that they may be called directly. This is also true on SGI Linux systems. However, the OpenGL application binary interface (ABI) for Linux does not guarantee that extension functions or core functions beyond the set of functions defined in OpenGL 1.2 and GLX 1.3 can be called statically on all Linux environments. This is because the OpenGL library, which defines static entry points, and the OpenGL hardware drivers, which define extensions, may come from different sources and, therefore, not always be compatible with each other. This is also true on Microsoft Windows systems.

As a result, the following applications must access extension functions and newer core functions, those beyond OpenGL 1.2 and GLX 1.3, dynamically at run time:

- Applications written to be portable to Linux systems by other vendors
- Applications written to be portable to Microsoft Windows systems
- Applications originally written on Linux systems by other vendors or Microsoft Windows systems and ported to SGI Linux systems

The GLX_ARB_get_proc_address extension allows dynamic access to these functions at run time by providing the **glXGetProcAddressARB()** function.

The glXGetProcAddressARB() Function

Function **glXGetProcAddressARB()** is called with the name of another OpenGL or GLX extension function or a newer core function and has the following format:

```
void (*glXGetProcAddress(const GLubyte *procname))(void)
```

The value *procname* is a string such as “glIsObjectBufferATI” or “glCompressedTexImage2DARB”. If the OpenGL or GLX function corresponding to *procname* exists, **glXGetProcAddressARB()** returns a function pointer to the corresponding function. Because the signatures of extension functions differ, the type of the pointer returned by **glXGetProcAddressARB()** is the generic (void (*)(void)). The pointer must be mapped to an appropriate function pointer type corresponding to the extension and then used to call the extension function when required.

The standard headers GL/glext.h and GL/glxext.h define, respectively, OpenGL and GLX interfaces (formats and token values) for extensions and newer core functions. In addition to defining formats, the headers also define C typedefs for these functions to ease the process of storing return values from **glXGetProcAddressARB()**. The following is a consistent convention for deriving a typedef for such a function:

1. Convert the name of the function to upper case.
2. Prefix the result with “PFNGL” (meaning “pointer to GL function”).
3. Suffix the result with “PROC”.

For example, consider the following extension function:

```
GLboolean glIsObjectBufferATI(GLuint buffer);
```

Its corresponding typedef in glext.h would be the following:

```
typedef GLboolean (*PFNGLISOBJECTBUFFERATIPROC)(GLuint buffer);
```

The typedef names for GLX extension functions are defined similarly, but using the prefix “PFNGLX” instead of “PFNGL”.

Example 5-2 shows how to query and use an extension function pointer. The example uses the **glIsObjectBufferATI function()**, which is part of the GL_ATI_vertex_array_object extension, supported on Onyx4 and Silicon Graphics Prism systems. The example assumes that the application has already checked the GL_EXTENSIONS string to confirm that the extension is supported and that all references to functions, typedefs, or tokens used by the extension are wrapped in a #ifdef GL_ATI_vertex_array_object/#endif block so that the application using this code compiles correctly on platforms not supporting the extension. For clarity, these compile- and run-time checks are not included in the example.

Example 5-2 Querying Extension Function Pointers

```
/* Declare global variable containing the extension function pointer */
PFNGLISOBJECTBUFFERATIPROC IsObjectBufferATI = NULL;

/* Query the function pointer */
IsObjectBufferATI = (PFNGLISOBJECTBUFFERATIPROC)
    glXGetProcAddressARB("glIsObjectBufferATI");

/* This should never happen if the extension is supported;
 * but sanity check anyway, for robustness. */
if (IsObjectBufferATI == NULL) {
    error("Cannot obtain extension function pointer.");
}

...

/* Later in the program, call the extension function as needed */
GLuint buffer = bufferID; /* A buffer ID to be queried */

/* Equivalent to calling
 *  if (glIsObjectBufferATI(buffer) == GL_TRUE) { ...
 */
if ((*IsObjectBufferATI)(buffer) == GL_TRUE) {
    /* buffer is indeed a vertex array buffer ID */
}
```

Note: Calling **glXGetProcAddressARB()** is an expensive operation. Do not call it every time an extension is to be called. Instead, query function pointers once after creating a context and cache the resulting pointers for future use.

Extension Wrapper Libraries and Portability Notes

Using the GLX_ARB_get_proc_address extension directly, as described in Example 5-2, can be tedious and intrusive on application code by causing many GL function calls to be performed indirectly through function pointers. Instead, use one of the many available open source extension wrapper libraries, which hide most of the details.

SGI does not currently recommend or support a specific wrapper library, because there are many popular libraries and they are frequently updated to keep track of new vendor and ARB-approved OpenGL extensions. Consult the the developer and support forums

area of the OpenGL website, http://www.opengl.org/ (which is also a good place to look for information about many other OpenGL programming topics).

As a starting point, consider the following extension wrapper libraries:

Wrapper Library	Website
GLEW	(The OpenGL Extension Wrangler library) http://glew.sourceforge.net/
`GLee`	(The OpenGL Easy Extension library) http://elf-stone.com/downloads.php#GLee
`extglgen`	http://trenki.50free.org/extgl/
OglExt	http://www.julius.caesar.de/oglext/

Note that most of these libraries run on Microsoft Windows as well as Linux (and sometimes other operating systems with OpenGL support as well).

Finding Information About Extensions

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

Man Pages

For the most up-to-date information on extensions, see the following man pages:

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

`glXintro` Information on GLX extensions.

Note that individual OpenGL man pages have a `MACHINE DEPENDENCIES` section that lists the systems on which certain extension functions or options are implemented.

Multisampling is supported on all current Silicon Graphics systems with the exception of Fuel, Tezro, and InfinitePerformance systems. Currently, it can be used with windows or pixel buffers 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 on IRIX systems 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://toolbox.sgi.com/.

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 details on any of the extensions, search for its specification in the OpenGL Extenstion Registry, http://oss.sgi.com/projects/ogl-sample/registry/.

Resource Control Extensions

This chapter describes 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, including the following:

- Introductory information—See the `glxintro` man 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" on page xl for bibliographical information.

This chapter explains how to use extensions to GLX. The following extensions are presented in alphabetical order:

The following sections describe extensions that are experimental:

Using OpenGL in an X Window System environment is described 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” on page 98 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()** does not create a new XID. It merely makes an existing XID available to the importing client. The XID goes away when the creating client drops its connection or the ID is explicitly deleted. The object goes away when the XID goes away and the context is not current to any thread.

- To free the client-side part of a GLX context that was created with **glXImportContextEXT()**, call **glXFreeContextEXT()**:

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

  glXFreeContextEXT() does not free the server-side context information or the XID associated with the server-side context.

Retrieving Display and Context Information

Use the extension to retrieve the display of the current context or other information about the context, as follows:

- To retrieve the current display associated with the current context, call **glXGetCurrentDisplayEXT()**, which has the following format:

  ```
  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()** does not 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

The EXT_import_context extension introduces the following new functions:

- **glXGetCurrentDisplayEXT()**
- **glXGetContextIDEXT()**
- **glXImportContextEXT()**
- **glXFreeContextEXT()**
- **glXQueryContextInfoEXT()**

SGI_make_current_read—The Make Current Read Extension

Note: The functionality of SGI_make_current_read was promoted to a standard part of GLX 1.3. For new applications, use the GLX 1.3 **glXMakeContextCurrent()** and **glXGetCurrentReadDrawable()** functions instead of this 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 )
```

The variable items are defined as follows:

dpy	Specifies the connection to the X server.
draw	Specifies a GLX drawable that receives the results of OpenGL drawing operations.
read	Specifies a GLX drawable that provides pixels for **glReadPixels()** and **glCopyPixels()** operations.
gc	Specifies 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 **glXMakeCurrentSGI()**.The function **glXMakeCurrentReadSGI()** lets you attach a GLX context to two drawables: you draw to the first one and the second serves as a source for pixel data.

In effect, the following calls are equivalent:

`glXMakeCurrentSGI(`*context, win)*
`glXMakeCurrentReadSGI(`*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()**.

The following 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 do not need to match. Some implementations may impose additional constraints, such as requiring that the color component resolution of both drawables be the same. In such cases, a `BadMatch X protocol` error will be generated.

Retrieving the Current Drawable's Name

The function **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

The SGI_make_current_read extension introduces the following functions:

- **glXMakeCurrentReadSGI()**
- **glXGetCurrentReadDrawableSGI()**

EXT_visual_info—The Visual Info Extension

Note: The functionality of EXT_visual_info was promoted to a standard part of GLX 1.3, which is supported on all current Silicon Graphics visualization systems. For new applications, use the GLX 1.3 **glXChooseFBConfig()** function and select framebuffer configurations based on the `GLX_X_VISUAL_TYPE` attribute instead of the `GLX_X_VISUAL_TYPE_EXT` attribute defined by this 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

Table 6-2 Heuristics for Visual Selection **(continued)**

If...	And GLX_X_VISUAL_TYPE_EXT is...	The result is...
	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. For more details, see the man page for **glXChooseVisual()**.

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

Do not 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" on page 65 presents an example program that uses a transparent visual for the overlay window.

EXT_visual_rating—The Visual Rating Extension

Note: The functionality of EXT_visual_rating was promoted to a standard part of GLX 1.3, which is supported on all current Silicon Graphics visualization systems. For new applications, use the GLX 1.3 **glXChooseFBConfig()** function and select framebuffer configurations based on the GLX_CONFIG_CAVEAT attribute instead of the GLX_VISUAL_CAVEAT_EXT attribute defined by this extension.

The visual rating extension, EXT_visual_rating, allows servers to export visuals with improved features or image quality but with lower performance or greater system burden. The extension allows this 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. 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`. The function **glXGetConfig()** returns the rating of the visual in the parameter *value*, which will be either `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_fbconfig—The Framebuffer Configuration Extension

Note: The functionality of SGIX_fbconfig was promoted to a standard part of GLX 1.3, which is supported on all current Silicon Graphics visualization systems. For new applications, use the GLX 1.3 equivalent functions and tokens instead of this extension. For a description of framebuffer configurations in GLX 1.3, see section "Using Visuals and Framebuffer Configurations" on page 71. Since the GLX 1.3 features are similar to this extension, the lengthy description is not repeated here.

SGIX_pbuffer—The Pixel Buffer Extension

Note: The functionality of SGIX_pbuffer was promoted to a standard part of GLX 1.3, which is supported on all current Silicon Graphics visualization systems. For new applications, use the GLX 1.3 equivalent functions and tokens instead of this extension. For a description of pixel buffers in GLX 1.3, see section "Using Pixel Buffers" on page 90. Since the GLX 1.3 features are similar to this extension, the lengthy description is not repeated here.

Vertex Processing Extensions

This chapter describes how to use the following OpenGL vertex processing extensions:

The following groups of obsolete (legacy) vertex processing extensions are also briefly described:

The legacy extensions are supported for compatibility and are not fully documented in this guide.

ARB_vertex_buffer_object—The Vertex Buffer Object Extension

The ARB_vertex_buffer_object extension allows applications to store buffers containing application-defined data in graphics memory and to draw vertex arrays using data contained in those buffers, instead of the usual vertex array usage where array data is taken from application memory.

Why Use Buffer Objects?

When drawing vertex arrays using unextended OpenGL 1.3, all data in the arrays must be transferred from application memory to the graphics processor. In Onyx4 and Silicon Graphics Prism systems (as well as all other modern graphics systems), the bandwidth between application memory and the graphics processor (typically over an interface like PCI-X or AGP) is substantially lower than the bandwidth between the graphics processor and its own local graphics memory. Therefore, when drawing vertex array data repeatedly with no changes or only small changes relative to the size of the arrays, substantial performance increases can be realized by storing vertex arrays in graphics memory. It is impossible to reach the maximum vertex transformation rates supported by the graphics processor unless vertex data is being supplied from graphics memory.

This extension provides an explicit mechanism for creating and managing data buffers in graphics memory by defining portions of those buffers as vertex arrays and drawing vertices using those arrays.

Alternatives to Buffer Objects

In the past, optimization advice often included the use of OpenGL display lists to encapsulate drawing commands. Display lists can also be stored in graphics memory and provide similar performance benefits. However, display lists cannot be modified once they are created; even the simplest change to a list requires destroying and re-creating its entire contents. Also, it is considerably more difficult for the graphics library to recognize and optimize display lists, because they can contain arbitrary sequences of OpenGL commands, not just array data.

While Onyx4 and Silicon Graphics Prism systems do perform display list optimizations, new applications should use buffer objects if possible. Buffer objects are more easily optimized, and individual elements of a buffer object can be modified without needing to re-create the entire buffer in graphics memory.

Another approach to high-performance drawing operations used in the past is for the application to hint to the graphics library that its vertex arrays will not be modified for some period of time by locking portions of the currently bound vertex arrays (see section “EXT_compiled_vertex_array—The Compiled Vertex Array Extension” on page 137). Locking allows the graphics library to copy vertex array data into graphics memory for the duration of the lock. However, any changes to vertex array data requires the expensive operations of unlocking, changing, and re-locking the array. Also, only a single

set of vertex arrays can be locked at a time; therefore, if multiple arrays are used for drawing, the performance benefits of locking are lost.

While Onyx4 and Silicon Graphics Prism systems do support locking vertex arrays, new applications should use buffer objects if possible. Multiple buffer objects can be defined and switched without swapping buffer data out of graphics memory and, as just described, individual elements of buffer objects can easily be modified.

Disadvantages of Buffer Objects

While buffer objects are the easiest and most reliable way to achieve maximum geometry throughput, graphics memory is usually a much more limited resource than application memory. Typically graphics processors have only 256–512 MB of graphics memory, and that memory must be shared among the framebuffer, texture, display lists, and buffer object storage.

If an application's use of graphics memory exceeds the amount physically present in the system, data may be automatically swapped out when not in use. This can result in greatly reduced performance and, in extreme cases, may result in applications terminating due to excessive graphics memory use. Examples where such situations are likely to arise include applications using many 2D image textures, using large 3D textures for volume rendering, or using large vertex arrays for drawing high-complexity models. In such cases, applications can achieve better performance by managing the swapping of texture and buffer data into graphics memory manually instead of relying on the automatic algorithms supported within the graphics library. However, such buffer management can be difficult to tune. A recommended alternative is to use higher-level scene graph APIs built on OpenGL, like OpenGL Performer and OpenGL Volumizer. These software layers are optimized to achieve maximum performance on Silicon Graphics systems while still supporting very large datasets.

Using Buffer Objects

As shown in the following code lines, buffer objects are represented by object names (of type `GLuint`) which are managed in exactly the same fashion as texture and display list names with routines for allocating unused buffer object names, deleting named buffer objects, and testing if a name refers to a valid buffer object:

```
void glGenBuffersARB(GLsizei n, GLuint *buffers);
void glDeleteBuffersARB(GLsizei n, const GLuint *buffers);
GLboolean glIsBufferARB(GLuint buffer);
```

Note that when deleting a buffer object with **glDeleteBuffersARB()**, all data in graphics memory associated with that buffer object will be freed as well. Because graphics memory is usually a scarce resource compared to application memory, it is important to delete buffer objects if they are no longer needed or to reuse the memory associated with buffer objects.

Defining Buffer Objects

Once a buffer object name has been obtained from **glGenBuffers()**, the corresponding buffer object can be created by making the following call:

```
void glBindBufferARB(GLenum target, GLuint buffer);
```

The argument *buffer* is the buffer object name, and *target* is either `GL_ARRAY_BUFFER_ARB` (for vertex array data) or `GL_ELEMENT_ARRAY_BUFFER_ARB` (for array index data). The newly created buffer object is initially defined with a size of zero.

You can also use **glBindBufferARB()** to bind an existing buffer object. If the bind is successful, no change is made to the state of the newly bound buffer object and any previous binding to target is broken.

While a buffer object is bound, operations on the target to which it is bound affect that object, and queries of the target return information about that object.

Initially, the reserved buffer object name `0` is bound to each of `GL_ARRAY_BUFFER_ARB` and `GL_ELEMENT_ARRAY_BUFFER_ARB`. However, there is no buffer object corresponding to the name `0`, and any attempt to operate on or query the `GL_ARRAY_BUFFER_ARB` or `GL_ELEMENT_ARRAY_BUFFER_ARB` target when it is bound to zero will generate errors. This is because binding to zero is used to indicate that normal vertex array behavior should apply, as described further later in section “Using Buffer Objects as Vertex Array Sources” on page 130.

Defining and Editing Buffer Object Contents

Buffer objects contain the same data that a normal OpenGL vertex array would contain, and the data is laid out in the same fashion. However, instead of simply providing a pointer to vertex array data in application memory, the contents of buffer objects must be explicitly defined.

Once a valid buffer object has been bound, define its contents by making the following call:

```
void glBufferDataARB(GLenum target, GLsizeiptrARB size, const void *data,
                     GLenum usage);
```

target If the buffer contents are to be used for vertex array data (for example, vertices, normals, texture coordinates, etc.), then *target* must be `GL_ARRAY_BUFFER_ARB`. If the contents are to be used for vertex index data (for example, indices into vertex array data), then target must be `GL_ELEMENT_ARRAY_BUFFER_ARB`. This target is described further later in the section "Using Buffer Objects as Vertex Array Sources" on page 130.

data A pointer to the buffer data in application memory. The argument data may be `NULL`, in which case the buffer object size is set as specified, but its contents remain undefined.

size The length of *data* in basic machine units (bytes). The type of *size* is the new C type `GLsizeiptrARB`. This type is an unsigned integer type guaranteed to be large enough to represent the largest possible object in application memory.

usage Provides a hint as to the expected usage pattern of the buffer being defined. The following are the valid usage hints:

`GL_STREAM_DRAW_ARB`
: Buffer contents will be specified once by the application and used at most a few times as the source of a drawing command.

`GL_STREAM_READ_ARB`
: Buffer contents will be specified once by reading data from OpenGL and queried at most a few times by the application.

`GL_STREAM_COPY_ARB`
: Buffer contents will be specified once by reading data from OpenGL and used at most a few times as the source of a drawing command.

`GL_STATIC_DRAW_ARB`
: Buffer contents will be specified once by the application and used many times as the source for drawing commands.

`GL_STATIC_READ_ARB`
: Buffer contents will be specified once by reading data from OpenGL and queried many times by the application.

`GL_STATIC_COPY_ARB`
: Buffer contents will be specified once by reading data from OpenGL and used many times as the source for drawing commands.

`GL_DYNAMIC_DRAW_ARB`
: Buffer contents will be respecified repeatedly by the application and used many times as the source for drawing commands.

`GL_DYNAMIC_READ_ARB`
: Buffer contents will be respecified repeatedly by reading data from OpenGL and queried many times by the application.

`GL_DYNAMIC_COPY_ARB`
: Buffer contents will be respecified repeatedly by reading data from OpenGL and used many times as the source for drawing commands.

The most common usage patterns for buffer objects being used as vertex array or element sources are the following:

`GL_STATIC_DRAW_ARB`
: Used for unchanging objects. This usage is similar to creating display lists that will be called many times.

`GL_DYNAMIC_DRAW_ARB`
: Used for objects whose contents may be edited repeatedly.

Many of the usage patterns are only expected to be relevant for future extensions built on ARB_vertex_buffer_object that use the same buffer object mechanism for other purposes, such as pixel or video data.

To edit (update) the contents of an existing buffer object by changing only part of the buffer contents, make the following call:

```
void glBufferSubDataARB(GLenum target, GLintptrARB offset, GLsizeiptrARB
                        size, const void *data);
```

The arguments *target*, *data*, and *size* specify the buffer object target to be affected, a pointer to the updated data block in application memory, and the length of the data block to replace in the buffer in the same fashion as the corresponding parameters of **glBufferDataARB()**.

The argument *offset* specifies the start of the range of data to replace in the buffer object in basic machine units relative to the beginning of the buffer being modified. The type of *offset* is the new C type `GLintptrARB`. This type is an integer type guaranteed to be large enough to represent the largest possible offset to an element of a buffer in application memory.

Elements *offset* through (*offset* + *size* – 1) in the buffer object bound to *target* are replaced by the corresponding elements in application memory starting at *data*. An error is generated if *offset* is less than zero, or if (*offset* + *size*) is greater than the size of the buffer object.

Mapping Buffer Objects to Application Memory

An alternate method for editing buffer objects is to map them into application memory by making the following call:

```
void *glMapBufferARB(GLenum target, GLenum access);
```

If the buffer object bound to *target* can be successfully mapped, a pointer to the buffer contents is returned; otherwise, a `GL_OUT_OF_MEMORY` error will be generated.

The argument *access* must be one of `GL_READ_ONLY_ARB`, `GL_WRITE_ONLY_ARB`, or `GL_READ_WRITE_ARB`. It specifies which operations may be performed on the buffer while it is mapped. The most common access pattern for buffer objects being used as vertex array sources is `GL_WRITE_ONLY_ARB`. It indicates that small parts of the buffer may be updated, but nothing will be read from the buffer.

While a buffer object is mapped, no OpenGL operations may refer to the mapped data either by issuing drawing commands that would refer to data in the mapped buffer object or by passing pointers within the mapped region to other OpenGL commands. Also, **glBufferSubData()** may not be called while the corresponding buffer object is mapped.

After modifying mapped buffer object contents and before using that buffer object as a source or sink for OpenGL, unmap the buffer object by making the following call:

```
GLboolean glUnmapBufferARB(GLenum target);
```

If **glUnmapBufferARB()** returns GL_FALSE, it indicates that values in the buffer object's data have become corrupted (usually as the result of a screen resolution change or another event that affects graphics memory). In this case, the buffer object contents are undefined.

Note: Mapping buffer objects into application memory may be a very inefficient way to modify their contents especially when performing indirect rendering, and such mapping has several possible failure modes caused by external events such as resolution changes. If possible, use **glBufferSubData()** to update buffer contents instead.

Using Buffer Objects as Vertex Array Sources

Once you create a buffer object and define its contents, you can use it as a source for array drawing operations. When any of the commands defining an array pointer (including those in the following list) is called **while a buffer object is bound**, the interpretation of the pointer argument to that command is changed:

- **glColorPointer()**
- **glEdgeFlagPointer()**
- **glIndexPointer()**
- **glNormalPointer()**
- **glTexCoordPointer()**
- **glVertexPointer()**
- **glFogCoordPointerEXT()**, if the EXT_fog_coord extension is supported
- **glSecondaryColorPointerEXT()**, if the EXT_secondary_color extension is supported
- **glVertexAttribPointerARB()**, if the ARB_vertex_program extension is supported
- **glWeightPointerARB()**, if the ARB_vertex_blend extension is supported

Instead of being interpreted as a pointer to data in application memory, the pointer is interpreted as **an offset within the currently bound buffer object**.

After defining a particular array pointer in this fashion and when the corresponding array is enabled, all vertex array drawing operations (for example, those in the following

list) will read data from the corresponding buffer object instead of from application memory:

- **glArrayElement()**
- **glDrawArrays()**
- **glDrawElements()**
- **glDrawRangeElements()**
- **glMultiDrawArrays()**
- **glMultiDrawElementsEXT()**

Once an array pointer is defined as an offset within a buffer object, the buffer object may be unbound, but the array pointer will continue to refer to that buffer object until it is redefined. This allows different array pointers to refer to different buffer objects, as well as to application memory. However, for maximum performance, all enabled array pointers should refer to buffer objects, both because any access to application memory while drawing is likely to limit performance due to bandwidth constraints and the complexity of mixing arrays from application and buffer object memory may throw the OpenGL implementation onto a slower and more complex code path.

When specifying array pointers as offsets within buffer objects, the application must convert an integer offset, expressed in basic machine units into a pointer argument. For this purpose, it is useful to define a macro like the following:

```
#define BUFFER_OFFSET(offset) ((char *)NULL + (offset))
```

For example, suppose that the bound buffer object contains an array of packed 3-component, floating point normal data and you wish to set the normal pointer to the 64th element of this array. In this case, the offset in basic machine units would be 64 * 3 * **sizeof(**`GLfloat`**)**. Therefore, you would make the following call:

```
glNormalPointer(3, GL_FLOAT, 0,
                BUFFER_OFFSET(64 * 3 * sizeof(GLfloat)));
```

Using Buffer Objects as Array Indices

In addition to storing vertex array *data* in buffer objects, array *indices* may also be stored. These indices are normally specified as pointer arguments to the array drawing commands **glDrawElements()**, **glDrawRangeElements()**, and (if the EXT_multi_draw_arrays extension is supported) **glMultiDrawElementsEXT()**. By storing both array data and array indices in buffer objects, indexed drawing operations

do not need to refer to application memory ever once they are set up. This enables maximum performance.

Array indices in buffer objects are defined using the same calls as for array data—for example, **glBindBufferARB()**, **glBufferDataARB()**, etc. However, the target `GL_ELEMENT_ARRAY_BUFFER_ARB` must be used for indices instead of `GL_ARRAY_BUFFER_ARB`.

In the same fashion as the array pointer calls, if **glDrawElements()** or **glDrawRangeElements()** is called while a buffer object is bound to `GL_ELEMENT_ARRAY_BUFFER_ARB`, the indices argument to these calls is interpreted as an offset into the buffer object, rather than a pointer to index data in application memory. If **glMultiDrawElementsEXT()** is called, the *indices* argument is still interpreted as a pointer into application memory; however, the contents of the memory located at that pointer are then interpreted as an array of offsets into the buffer object, rather than an array of pointers into application memory.

Querying Data in Buffer Objects

To query part or all of the contents of a buffer object, make the following call:

```
void glGetBufferSubDataARB(GLenum target, GLintptrARB offset,
                           GLsizeiptrARB size, void *data);
```

The argumemts *target*, *offset*, and *size* have the same meaning as the corresponding arguments of **glBufferSubDataARB()**; they specify the target to be queried and the range of data within the buffer object bound to that target to return. The returned data is copied to the region of application memory referenced by *data*.

Buffer object contents may not be queried while an object is mapped; calls to **glGetBufferSubDataARB()** will generate a `GL_INVALID_OPERATION` error in this case.

Sample Code

The following code fragment defines two buffer objects, fills them with data interpreted respectively as vertex coordinates and vertex colors, and draws a triangle using the data contained in the buffer objects.

```
#define BUFFER_OFFSET(offset) ((char *)NULL + (offset))

/* Vertex coordinate and color data to place in buffer objects */
GLfloat vertexData[] = { -1.0,  1.0,  0.0,
```

```
                          -1.0, -1.0,  0.0,
                           1.0, -1.0,  0.0 };
GLfloat colorData[] = {  0.0,  0.0,  0.0,
                         1.0,  0.0,  0.0,
                         1.0,  1.0,  0.0 };

/* Names of the vertex and color buffer objects */
GLuint vertexBuffer, colorBuffer;

/* Generate two unused buffer object names */
glGenBuffersARB(1, &vertexBuffer);
glGenBuffersARB(1, &colorBuffer);

/* Bind the first buffer object and fill it with vertex data */
glBindBufferARB(GL_ARRAY_BUFFER, vertexBuffer);
glBufferDataARB(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData,
GL_STATIC_DRAW);

/* Bind the second buffer object and fill it with color data */
glBindBufferARB(GL_ARRAY_BUFFER, colorBuffer);
glBufferDataARB(GL_ARRAY_BUFFER, sizeof(colorData), colorData,
GL_STATIC_DRAW);

/* Enable vertex and color arrays for drawing */
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

/* Set the vertex array pointer to the start of the vertex buffer
object */
glBindBufferARB(GL_ARRAY_BUFFER, vertexBuffer);
glVertexPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));

/* Set the color array pointer to the start of the color buffer object
*/
glBindBufferARB(GL_ARRAY_BUFFER, colorBuffer);
glColorPointer(3, GL_FLOAT, 0, BUFFER_OFFSET(0));

/* Unbind the array buffer target, since all enabled array
 * pointers have now been set.
 */
glBindBufferARB(GL_ARRAY_BUFFER, 0);

/*
 * Everything up to this point is initialization. Now the application
 * can enter its drawing loop.
```

```
 */

while (!drawingLoopDone()) {
    /* Perform input and per-loop processing, if required */
    doLoopProcessing();

    /* Draw the triangle defined by the vertex and color buffer objects
*/
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 3);
}

/*
 * When the drawing loop is complete, buffer objects should be deleted.
 */

/* Disable the vertex and color arrays */
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);

/* Free data contained in the buffer objects, and delete the objects */
glDeleteBuffersARB(1, vertexBuffer);
glDeleteBuffersARB(1, colorBuffer);
```

New Functions

The ARB_vertex_buffer_object extension introduces the following functions:

- **glBindBufferARB()**
- **glBufferDataARB()**
- **glBufferSubDataARB()**
- **glDeleteBuffersARB()**
- **glGenBuffersARB()**
- **glGetBufferSubDataARB()**
- **glIsBufferARB()**
- **glMapBufferARB()**
- **glUnmapBufferARB()**

ARB_window_pos—The Window-Space Raster Position Extension

The ARB_window_pos extension provides a set of functions to directly set the raster position in window coordinates. This extension bypasses the model-view and projection matrices and the viewport-to-window mapping.

Why Use the Window-Space Raster Position Extension?

When drawing two-dimensional geometry, applications often want to have pixel-precise control of where pixels are drawn on the screen. Normally when specifying the current raster position, the raster position specified by the application is treated in the same fashion as a vertex: it is transformed by the model-view and projection matrices and then sent through the viewport-to-window mapping to arrive at a window-space raster position.

While it is possible to set the raster position to a specific window-space location using the conventional mechanism, doing so requires careful setup of the transformation matrices and viewport mapping. Also, if the projected window-space raster position is outside the window bounds, it may be marked invalid so that nothing will be drawn by **glDrawPixels()**, even though this effect may be desirable (for drawing pixel images that are partially outside the window but whose visible regions are still drawn).

This extension introduces a mechanism for directly setting the raster position in window-space coordinates and ensuring that the resulting raster position will always be valid even if it is outside the window.

Using the Window-Space Raster Position Extenstion

The current raster position may be defined in window space with any of the following calls:

```
void glWindowPos2sARB(GLshort x, GLshort y);
void glWindowPos2iARB(GLint x, GLint y);
void glWindowPos2fARB(GLfloat x, GLfloat y);
void glWindowPos2dARB(GLdouble x, GLdouble y);
void glWindowPos3sARB(GLshort x, GLshort y, GLshort z);
void glWindowPos3iARB(GLint x, GLint y, GLint z);
void glWindowPos3fARB(GLfloat x, GLfloat y, GLfloat z);
void glWindowPos3dARB(GLdouble x, GLdouble y, GLdouble z);
```

In the **glWindowPos2*()** forms of this call, only the x and y raster position coordinates are specified, and raster position z is always set to zero. In the **glWindowPos3*()** forms, x, y, and z are all specified.

The following are the vector forms of these calls;

```
void glWindowPos2svARB(const GLshort *pos);
void glWindowPos2ivARB(const GLint *pos);
void glWindowPos2fvARB(const GLfloat *pos);
void glWindowPos2dvARB(const GLdouble *pos);
void glWindowPos3svARB(const GLshort *pos);
void glWindowPos3ivARB(const GLint *pos);
void glWindowPos3fvARB(const GLfloat *pos);
void glWindowPos3dvARB(const GLdouble *pos);
```

In the **glWindowPos2*vARB()** forms of this call, the argument is a pointer to a two-element vector specifying x and y, and the raster position z is always set to zero. In the **glWindowPos3*vARB()** forms, the argument is a pointer to a three-element vector specifying x, y, and z.

For all forms of **glWindowPos*()**, associated data (raster color, texture coordinates, etc.) for the current raster position is taken from the current state values in the same fashion as for **glRasterPos*()**. However, lighting, texture coordinate generation, and clipping are not performed by **glWindowPos*()**.

New Functions

The ARB_window_pos extesnion introduces the 16 functions listed in preceding section.

EXT_clip_volume_hint—The Clip Volume Hint Extension

The EXT_clip_volume_hint extension provides a mechanism for applications to indicate that they do not require clip volume clipping for primitives. It allows applications to maximize performance in situations where they know that clipping is unnecessary.

Why Use Clip Volume Hints?

Clipping geometry to the clip volume can decrease performance, and is not always needed. In many situations, applications can determine that part or all of the geometry being rendered lies entirely inside the clip volume; in other words, that such geometry will never be clipped. This is typically done by testing bounding boxes around application geometry against the clip volume. While such tests might in principle be done using OpenGL features such as the NV_occlusion_query extension, it is usually best to simply compare bounding boxes against the plane equations defining the clip volume entirely in the application code.

Using Clip Volume Hints

To hint that clip volume clipping does not need to be performed, call **glHint()** with a target of `CLIP_VOLUME_CLIPPING_HINT_EXT` and a mode of `GL_FASTEST`. To hint that clip volume clipping must be performed, use a mode of `GL_NICEST` instead.

As with all hints, the clip volume hint is only an indication and the OpenGL implementation may not respect the hint when set to `GL_FASTEST`. However, if large amounts of geometry can easily be tested to confirm that they need not be clipped, then there may be performance gains in using the hint particularly when using multiple user-defined clipping planes.

EXT_compiled_vertex_array—The Compiled Vertex Array Extension

The EXT_compiled_vertex_array extension defines an interface which allows static (unchanging) vertex arrays in application memory to be cached, pre-transformed, or pre-compiled.

Why Use Compiled Vertex Arrays?

Compiled vertex arrays may be used to cache the transformed results of array data for reuse by several **glDrawArrays()**, **glArrayElement()**, or **glDrawElements()** commands. For example, you might get better performance when drawing a large mesh of quadrilaterals one strip at a time, where each successive strip shares half its vertices with the previous strip. It also allows transferring array data to faster memory for more efficient processing.

Using compiled vertex arrays is an optimization technique that should be used only when porting old code that already uses client-side vertex arrays for drawing. Whenever possible in new applications, use buffer objects instead (see "ARB_vertex_buffer_object—The Vertex Buffer Object Extension" on page 123).

Compiled vertex arrays should be used only when executing multiple vertex array drawing commands that collectively refer multiple times to most of the elements in the locked range. The performance benefits of using compiled vertex arrays with very small vertex arrays (consequently, not reusing many elements) are unlikely to be worthwhile.

Using Compiled Vertex Arrays

To use compiled vertex arrays, follow these steps:

1. Identify the range of elements of the currently bound vertex arrays that may be reused in subsequent drawing operations
2. Make the following call:

   `void glLockArraysEXT(GLint` *first*`, GLsizei` *count*`);`

   The argument first specifies a starting element index and count specifies the number of elements to lock. Elements first through (*first* + *count* – 1) of **all** enabled vertex arrays will be locked.
3. Render geometry using **glDrawArrays()**, **glDrawElements()**, or other vertex array drawing commands.

   While vertex arrays are locked, changes made to array contents by an application may not be reflected in any vertex array drawing commands. Furthermore, vertex array drawing commands that refer to array elements outside the locked range have undefined results.
4. When finished drawing data in the locked ranges, make the following call:

   `void glUnlockArraysEXT(void);`

   This unlocks all arrays; subsequent changes to vertex arrays are properly reflected by drawing commands, and the restriction of drawing only elements within the locked range is lifted.

New Functions

The EXT_compiled_vertex_array extension introduces the following functions:

- **glLockArraysEXT()**
- **glUnlockArraysEXT()**

EXT_fog_coord—The Fog Coordinate Extension

The EXT_fog_coord extension introduces the fog coordinate, a new per-vertex attribute, which may be used in fog computation in place of the fragment's eye distance.

Why Use Fog Coordinates?

Normally, when fog is enabled, the fog factor computed for each fragment is based on the distance from the camera to the fragment. This distance is fed into one of three parameterized fog models (linear, exponential, or exponential-squared), as selected by parameters to **glFog*()**.

Fog models based only on fragment distance do not provide a level of control sufficient for effects such as patchy fog. By specifying arbitrary per-vertex values as input to the fog model rather than fragment distance, applications can produce more sophisticated and realistic fog models.

Using Fog Coordinates

To select use of either the fog coordinate or the fragment eye distance when computing fog, specify the fog coordinate source by making the following call:

```
glFogi(GL_FOG_COORDINATE_SOURCE_EXT, param);
```

If *param* is `GL_FOG_COORDINATE_EXT`, the fog coordinate is used in fog computations. If *param* is `GL_FRAGMENT_DEPTH_EXT`, the fragment eye distance is used. Initially fragment eye distance is used.

Fog coordinates are interpolated over primitives in the same fashion as colors, texture coordinates, and other vertex attributes. When drawing immediate-mode geometry, the current fog coordinate is specified by calling one of the following functions:

```
void glFogCoordfEXT(GLfloat coord);
void glFogCoorddEXT(GLdouble coord);
void glFogCoordfvEXT(GLfloat *coord);
void glFogCoorddvEXT(GLdouble *coord);
```

The fog coordinate may also be specified when drawing using vertex arrays. An array of per-vertex fog coordinates is defined by making the following call:

```
void glFogCoordPointerEXT(GLenum type, GLsizei stride, const GLvoid *ptr);
```

The argument *type* specifies the type of data in the array and must be either `GL_FLOAT` or `GL_DOUBLE`. The argument *stride* specifies the offset in basic machine units from one fog coordinate to the next in the array starting at *ptr*. As with other vertex array specification calls, a stride of zero indicates that fog coordinates are tightly packed in the array.

To enable or disable fog coordinates when drawing vertex arrays, call **glEnableClientState()** or **glDisableClientState()** with parameter `GL_FOG_COORDINATE_ARRAY_EXT`.

Querying the Fog Coordinate State

The current fog coordinate can be queried by calling **glGetFloatv()** with parameter name `GL_CURRENT_FOG_COORDINATE_EXT`. Parameters of the fog coordinate vertex array pointer can be queried by calling **glGetIntegerv()** with parameter name `GL_FOG_COORDINATE_ARRAY_TYPE_EXT` or `GL_FOG_COORDINATE_ARRAY_STRIDE_EXT` and calling **glGetPointerv()** with parameter name `GL_FOG_COORDINATE_ARRAY_POINTER_EXT`.

New Functions

The EXT_fog_coord extension introduces the following functions:

- **glFogCoordfEXT()**
- **glFogCoorddEXT()**
- **glFogCoordfvEXT()**

- **glFogCoorddvEXT()**
- **glFogCoordPointerEXT()**

EXT_multi_draw_arrays—The Multiple Draw Arrays Extension

The EXT_multi_draw_arrays extension defines two functions that allow multiple groups of primitives to be rendered from the same vertex arrays.

Why Use Multiple Draw Arrays?

When drawing many small, disjoint geometric primitives from a single set of vertex arrays, a separate call to **glDrawArrays()** or **glDrawElements()** is required for each primitive. This can be inefficient due to the setup required for each call. Using this extension, multiple disjoint ranges of vertex arrays can be drawn in a single call. This reduces the setup overhead and code complexity.

Using Multiple Draw Arrays

When drawing more than one range of data from a set of vertex arrays, where each such range is a contiguous group of elements in the arrays, make the following call:

```
void glMultiDrawArraysEXT(GLenum mode, const GLint *first,
                          const GLsizei *count, GLsizei primcount);
```

This is equivalent to the following multiple calls to **glDrawArrays()**:

```
for (int i = 0; i < primcount; i++) {
    if (count[i]) > 0)
        glDrawArrays(mode, first[i], count[i]);
}
```

When drawing more than one range of data, where each range is defined by a contiguous range of indices, make the following call:

```
void glMultiDrawElementsEXT(GLenum mode, const GLsizei *count,
                  GLenum type, const GLvoid **indices, GLsizei primcount);
```

This is equivalent to the following multiple calls to **glDrawElements()**;

```
for (int i = 0; i < primcount; i++) {
```

```
    if (count[i]) > 0)
        glDrawElements(mode, count[i], type, indices[i]);
}
```

The *i*th element of the count array is the number of array indices to draw, and the *i*th element of the index array is a pointer to the array indices. All indices must be of the same specified type.

New Functions

The EXT_multi_draw_arrays extension introduces the following functions:

- **glMultiDrawArraysEXT()**
- **glMultiDrawElementsEXT()**

EXT_secondary_color—The Secondary Color Extension

The EXT_secondary_color extension introduces the secondary color, a new per-vertex attribute. When lighting is disabled, the secondary color may be added to the color resulting from texturing. In unextended OpenGL 1.3, this color sum computation is only possible when lighting is enabled, and the secondary color used in this situation is based on the specular term of lighting equations rather than being explicitly defined by the application.

Why Use Secondary Color?

Many rendering algorithms use texture-based lighting computations rather than the builtin vertex lighting of OpenGL. While texture-based lighting is more difficult to specify, it supports arbitrary lighting models. In unextended OpenGL 1.3, the color sum hardware is not available to texture-based lighting. By introducting an explicit secondary color attribute, lighting effects such as non-textured specular highlights can easily be produced even when using texture-based lighting.

Using Secondary Color

To control the use of secondary color and color sum when OpenGL lighting is disabled, call **glEnable()** or **glDisable()** with parameter `GL_COLOR_SUM_EXT`.

Only the red, green, and blue components of the secondary color can be controlled; the alpha component is unused in the color sum and is assumed to be zero. Initially, the secondary color is (0,0,0).

Secondary color is interpolated over primitives in the same fashion as color. When drawing immediate-mode geometry, the current secondary color is specified by calling one of the following functions:

```
void glColor3bEXT(GLbyte red, GLbyte green, GLbyte blue);
void glColor3ubEXT(GLubyte red, GLubyte green, GLubyte blue);
void glColor3sEXT(GLshort red, GLshort green, GLshort blue);
void glColor3usEXT(GLushort red, GLushort green, GLushort blue);
void glColor3iEXT(GLint red, GLint green, GLint blue);
void glColor3uiEXT(GLuint red, GLuint green, GLuint blue);
void glColor3fEXT(GLfloat red, GLfloat green, GLfloat blue);
void glColor3dEXT(GLdouble red, GLdouble green, GLdouble blue);
void glColor3bvEXT(GLbyte *coords);
void glColor3ubvEXT(GLubyte *coords);
void glColor3svEXT(GLshort *coords);
void glColor3usvEXT(GLushort *coords);
void glColor3ivEXT(GLint *coords);
void glColor3uivEXT(GLuint *coords);
void glColor3fvEXT(GLfloat *coords);
void glColor3dvEXT(GLdouble *coords);
```

In the vector forms of these calls, coords is a three-element array containing red, green, and blue secondary color components in order. The data formats supported and interpretation of parameter values as color components are identical to the three-component glColor*() commands.

Secondary color may also be specified when drawing using vertex arrays. An array of per-vertex secondary colors is defined by making the following call:

```
void glSecondaryColorPointerEXT(GLint size, GLenum type, GLsizei stride,
                                const GLvoid *ptr);
```

The arguments are defined as follows:

size Specifies the number of components per color value and must always be 3.

type Specifies the type of data in the array and must be one of GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, or GL_DOUBLE.

stride Specifies the offset in basic machine units from one secondary color to the next in the array starting at *ptr*. As with other vertex array specification calls, a stride of zero indicates that secondary colors are tightly packed in the array.

To enable or disable secondary colors when drawing vertex arrays, call **glEnableClientState()** or **glDisableClientState()** with parameter GL_SECONDARY_COLOR_ARRAY_EXT.

Querying the Secondary Color State

The current secondary color can be queried by calling **glGetFloatv()** with parameter name GL_CURRENT_SECONDARY_COLOR_EXT. Parameters of the secondary color vertex array pointer can be queried by calling **glGetIntegerv()** with one of the following parameter names and calling **glGetPointerv()** with parameter name GL_SECONDARY_COLOR_ARRAY_POINTER_EXT:

- GL_SECONDARY_COLOR_ARRAY_SIZE_EXT
- GL_SECONDARY_COLOR_ARRAY_TYPE_EXT
- GL_SECONDARY_COLOR_ARRAY_STRIDE_EXT

New Functions

The EXT_secondary_color extension introduces the list of functions defined in section “Using Secondary Color” on page 143.

The Vertex Array Object Extensions (Legacy)

In addition to the ARB_vertex_buffer_object extension, Onyx4 and Silicon Graphics Prism systems also support the following set of ATI vendor extensions that were developed prior to ARB_vertex_buffer_object and were the basis on which ARB_vertex_buffer_object was specified:

- ATI_element_array
- ATI_map_object_buffer
- ATI_vertex_array_object
- ATI_vertex_attrib_array_object

Note: These four extensions are included only for support of legacy applications being ported from other platforms. They supply no functionality beyond that of ARB_vertex_buffer_object and are not as widely used. Whenever writing new code using buffer objects, always use the ARB extension.

Since these are legacy extensions, they are not documented in detail in this guide. The following table briefly describes each extension in terms of how it maps onto ARB_vertex_buffer_object:

ATI_vertex_array_object
: Defines the base functionality for creating array objects: defining usage modes and contents of array objects and defining specific vertex arrays as portions of array objects.

ATI_vertex_attrib_array_object
: Defines additional APIs for creating array objects that can contain vertex attribute data for use with the ARB_vertex_program and ARB_fragment_program extensions.

ATI_element_array
: Allows drawing array objects using arrays of indices also in array objects, analogous to the `ELEMENT_ARRAY_BUFFER_ARB` target supported by ARB_vertex_buffer_object.

ATI_map_object_buffer
: Allows mapping array objects into application memory, analogous to the **glMapBufferARB()** functionality of ARB_vertex_buffer_object.

New Functions

The legacy vertex array objects extensions introduce the following functions:

- **glArrayObjectATI()**
- **glDrawElementArrayATI()**
- **glDrawRangeElementArrayATI()**
- **glElementPointerATI()**
- **glFreeObjectBufferATI()**
- **glGetArrayObjectfvATI()**
- **glGetArrayObjectivATI()**
- **glGetObjectBufferfvATI()**
- **glGetObjectBufferivATI()**
- **glGetVariantArrayObjectfvATI()**
- **glGetVariantArrayObjectivATI()**
- **glGetVertexAttribArrayObjectfvATI()**
- **glGetVertexAttribArrayObjectivATI()**
- **glIsObjectBufferATI()**
- **glMapObjectBufferATI()**
- **glNewObjectBufferATI()**
- **glUnmapObjectBufferATI()**
- **glUpdateObjectBufferATI()**
- **glVariantArrayObjectATI()**
- **glVertexAttribArrayObjectATI()**

The Texture Coordinate Generation Extensions (Legacy)

There are two legacy texture coordinate generation extensions:

- EXT_texgen_reflection
- NV_texgen_reflection

The EXT_texgen_reflection extension provides two new texture coordinate generation modes that are useful in texture-based lighting and environment mapping. Differing only in the token names used, the NV_texgen_reflection provides identical functionality.

Note: The functionality defined by these extensions was later promoted into a standard part of OpenGL 1.3, and these extensions are included only for support of legacy applications being ported from other platforms. Whenever writing new code, always use the OpenGL 1.3 interface.

Since these are legacy extensions, they are not documented in detail here; only the mapping from the extension tokens to the OpenGL 1.3 tokens is defined.

EXT_texgen_reflection defines the following two new texture generation modes, according to the value of param to **glTexGeni()** when its *pname* argument is `GL_TEXTURE_GEN_MODE`:

- `GL_NORMAL_MAP_EXT`
- `GL_REFLECTION_MAP_EXT`

NV_texgen_reflection uses the following token names to define the same modes, respectively:

- `GL_NORMAL_MAP_NV`
- `GL_REFLECTION_MAP_NV`

In OpenGL 1.3, the mode defined by `GL_NORMAL_MAP_EXT` and `GL_NORMAL_MAP_NV` may instead be defined by `GL_NORMAL_MAP`. Likewise, the the mode defined by `GL_REFLECTION_MAP_EXT` and `GL_REFLECTION_MAP_NV` may instead be defined by `GL_REFLECTION_MAP`.

Texturing Extensions

This chapter explains how to use the following OpenGL texturing extensions:

This chapter also describe the following extensions that are experimental:

ATI_texture_env_combine3—New Texture Combiner Operations Extension

The OpenGL 1.3 core provides texture combiner operations. These operations are a powerful set of functions that can be applied at each texture unit and exceed the simpler OpenGL 1.0 texture environment functions, such as `ADD`. The extension ATI_texture_env_combine3 defines several additional combiner operations.

This section assumes familiarity with the basic texture combiner interface and only describes the new operations added by the extension.

Why Use Texture Combiners?

Texture combiners allow a greatly increased range of texturing functionality (per-pixel lighting, bump mapping, and other advanced rendering effects) while still using the OpenGL fixed-function pipeline. This extension increases that range even further compared to base OpenGL 1.3. For an even broader range of functionality, consider using fragment programs instead.

Using The New Texture Combiner Operations

When **glTexEnvi()** or **glTexEnvf()** is called with a parameter name of `GL_COMBINE_RGB` or `GL_COMBINE_ALPHA`, this extension allows the corresponding parameter to take on one of the following values:

- `GL_MODULATE_ADD_ATI`
- `GL_MODULATE_SIGNED_ADD_ATI`
- `GL_MODULATE_SUBTRACT_ATI`

Table 8-1 shows the texture functions corresponding to these operations.

Table 8-1 Additional Texture Combiner Operations

GL_COMBINE_RGB or GL_COMBINE_ALPHA Operation	Texture Function
GL_MODULATE_ADD_ATI	Arg0 * Arg2 + Arg1
GL_MODULATE_SIGNED_ADD_ATI	Arg0 * Arg2 + Arg1 – 0.5
GL_MODULATE_SUBTRACT_ATI	Arg0 * Arg2 – Arg1

In Table 8-1, Arg0, Arg1, and Arg2 represent values determined by the values set for GL_SOURCE(0,1,2)_(RGB,ALPHA) and GL_OPERAND(0,1,2)_(RGB,ALPHA) with **glTexEnv*()**. In addition to the values defined by OpenGL 1.3 (GL_TEXTURE, GL_CONSTANT, GL_PRIMARY_COLOR, and GL_PREVIOUS), this extension allows GL_SOURCE(0,1,2)_(RGB,ALPHA) to take on the values GL_ZERO and GL_ONE. In this case, the values generated for the corresponding Arg(0,1,2) are shown in Table 8-2 and Table 8-3.

Table 8-2 New Arguments for Texture Combiner Operations

GL_SOURCE(0,1,2)_RGB	GL_OPERAND(0,1,2)_RGB	Resulting Arg(0,1,2) RGB Value (for each component)
GL_ZERO	GL_SRC_COLOR	0
GL_ZERO	GL_ONE_MINUS_SRC_COLOR	1
GL_ZERO	GL_SRC_ALPHA	0
GL_ZERO	GL_ONE_MINUS_SRC_ALPHA	1
GL_ONE	GL_SRC_COLOR	1
GL_ONE	GL_ONE_MINUS_SRC_COLOR	0
GL_ONE	GL_SRC_ALPHA	1
GL_ONE	GL_ONE_MINUS_SRC_ALPHA	0

Table 8-3 New Arguments for Texture Combiner Operations (Alpha-Related)

GL_SOURCE(0,1,2)_ALPHA	GL_OPERAND(0,1,2)_ALPHA	Resulting Arg(0,1,2) Alpha Value
GL_ZERO	GL_SRC_ALPHA	0
GL_ZERO	GL_ONE_MINUS_SRC_ALPHA	1
GL_ONE	GL_SRC_ALPHA	1
GL_ONE	GL_ONE_MINUS_SRC_ALPHA	0

ATI_texture_float—The Floating Point Texture Extension

The ATI_texture_float extension defines new, sized texture internal formats with 32- and 16-bit floating point components. The 32-bit floating point components are stored in standard IEEE single-precision float format. The 16-bit floating point components have 1 sign bit, 5 exponent bits, and 10 mantissa bits. Floating point components are clamped to the limits of the range representable by their format.

Why Use Floating Point Textures?

Floating point textures support greatly increased numerical range and precision compared to fixed-point textures, which can only represent values in the range [0,1]. This is important for many purposes, such as high dynamic range imaging, performing general-purpose numerical computations in the graphics processor, and representing input data naturally without needing to scale and bias it to fit in the limited range of fixed-point textures.

Floating point textures are especially useful when using fragment shaders, where a much wider range of computations can be performed than in the fixed-function graphics pipeline.

Using Floating Point Textures

The new formats defined by this extension may be used as the `internalformat` parameter when specifying textures with one of the following:

- **glTexImage1D()**
- **glTexImage2D()**
- **glTexImage3D()**
- **glCopyTexImage1D()**
- **glCopyTexImage2D()**

The names of the new formats, the corresponding base internal format, and the precision of each component in a texture stored with those formats are shown in Table 8-4. In the table, "f32" means the component is stored as a 32-bit IEEE floating point number and "f16" means the component is stored as a 16-bit floating point number.

Table 8-4 New Floating Point Internal Formats for Textures

Sized Internal Format	Base Internal Format	Red Bits	Green Bits	Blue Bits	Alpha Bits	Lum Bits	Inten Bits
RGBA_FLOAT32_ATI	RGBA	f32	f32	f32	f32		
RGB_FLOAT32_ATI	RGB	f32	f32	f32			
ALPHA_FLOAT32_ATI	ALPHA				f32		
INTENSITY_FLOAT32_ATI	INTENSITY						f32
LUMINANCE_FLOAT32_ATI	LUMINANCE					f32	
LUMINANCE_ALPHA_FLOAT32_ATI	LUMINANCE_ALPHA				f32	f32	
RGBA_FLOAT16_ATI	RGBA	f16	f16	f16	f16		
RGB_FLOAT16_ATI	RGB	f16	f16	f16			
ALPHA_FLOAT16_ATI	ALPHA				f16		
INTENSITY_FLOAT16_ATI	INTENSITY						f16

Table 8-4 New Floating Point Internal Formats for Textures **(continued)**

Sized Internal Format	Base Internal Format	Red Bits	Green Bits	Blue Bits	Alpha Bits	Lum Bits	Inten Bits
LUMINANCE_FLOAT16_ATI	LUMINANCE					f16	
LUMINANCE_ALPHA_FLOAT16_ATI	LUMINANCE_ALPHA				f16	f16	

ATI_texture_mirror_once—The Texture Mirroring Extension

The ATI_texture_mirror_once extension introduces new texture coordinate wrap modes that effectively use a texture map twice as large as the specified texture image. The additional half of the new image is a mirror image of the original.

This behavior is similar to the GL_MIRRORED_REPEAT wrap mode of OpenGL 1.4, but mirroring is done only once rather than repeating. That is, input texture coordinates outside the range [–1,1] are clamped to this range. After clamping, values in the range [0,1] are used unchanged while values in the range [–1,0] are negated before sampling the texture.

The extension supports the following two wrap modes:

GL_MIRROR_CLAMP_ATI	Texture filtering may include texels from the texture border, like the core GL_CLAMP mode.
GL_MIRROR_CLAMP_TO_EDGE_ATI	Texture coordinates are clamped such that the texture filter never samples texture borders, like the core GL_CLAMP_TO_EDGE mode.

Why Use Texture Mirroring?

For textures that are symmetrical about one or more axes, texture mirroring reduces the amount of texture memory required by not storing the redundant symmetric portion of the texture. The choice of using GL_MIRRORED_REPEAT or the modes introduced by ATI_texture_mirror_once depends on whether or not an infinitely extended texture image is desired (as may be the case for synthetic textures used for backgrounds or high-frequency noise).

Using Texture Mirroring

To specify texture mirroring, call **glTexParameteri()** with the following parameter specifications:

target	`GL_TEXTURE_1D`,`GL_TEXTURE_2D`, or `GL_TEXTURE_3D`
pname	`GL_TEXTURE_WRAP_S`,`GL_TEXTURE_WRAP_T`, or `GL_TEXTURE_WRAP_R`
param	`GL_MIRROR_CLAMP_ATI` for mirroring or `GL_MIRROR_CLAMP_TO_EDGE_ATI` for mirroring without sampling texture borders

EXT_texture_compression_s3tc—The S3 Compressed Texture Format Extension

The EXT_texture_compression_s3tc extension builds on the compressed texture interface in core OpenGL by adding external and internal compressed formats in the popular S3TC formats. S3TC formats are also sometimes referred to as DXTC, in the Microsoft DirectX terminology. These formats are only supported for 2D textures.

Why Use S3TC Texture Formats?

Depending on the nature of the textures, compressed textures can provide dramatic savings in texture memory at a relatively small cost in texture quality. Natural imagery tends to compress with fewer detectable artifacts than synthetic images, but it is always important to test and make sure that compressed image quality is adequate, particularly in high-fidelity domains such as flight simulation.

The core OpenGL compressed texture interface allows compressing textures to a internal format whose exact nature is unspecified. This format may differ between OpenGL implementations, and potentially even between driver releases on the same platform. This variance makes it difficult to ensure that the quality and size of images so compressed are adequate. It is also difficult to store or transport compressed images without knowing their exact format.

S3TC overcomes these constraints by defining specific formats for compressed images of several types, and these formats may be used by other tools such as image viewers and

artwork creation applications. S3TC is a widely used informal standard for texture compression.

Using S3TC Texture Formats

This extension introduces four new compressed texture formats, with corresponding RGB or RGBA base formats as shown in Table 8-5.

Table 8-5 S3TC Compressed Formats and Corresponding Base Formats

Compressed Internal Format	Base Internal Format	Description
`GL_COMPRESSED_RGB_S3TC_DXT1_EXT`	`GL_RGB`	Each 4x4 block of texels consists of 64 bits of RGB image data.
`GL_COMPRESSED_RGBA_S3TC_DXT1_EXT`	`GL_RGBA`	Each 4x4 block of texels consists of 64 bits of RGB image data and minimal alpha information (1 bit/texel corresponding to `0.0` or `1.0`).
`GL_COMPRESSED_RGBA_S3TC_DXT3_EXT`	`GL_RGBA`	Each 4x4 block of texels consists of 64 bits of uncompressed alpha image data followed by 64 bits of compressed RGB image data.
`GL_COMPRESSED_RGBA_S3TC_DXT5_EXT`	`GL_RGBA`	Each 4x4 block of texels consists of 64 bits of compressed alpha image data followed by 64 bits of uncompressed RGB image data.

The new compressed formats may be used as the `internalformat` parameter of **glTexImage2D()**, **glCopyTexImage2D()**, and **glCompressedTexImage2D()** when specifying a 2D texture, and as the `format` parameter of **glCompressedTexSubImage2D()** when respecifying a part of a texture.

When specifying a texture in already-compressed S3TC format—for example, when calling **glCompressedTexImage2D()** or **glCompressedTexSubImage2D()**—the required format of the input image is fully defined by the extension specification for EXT_texture_compression_s3tc. The specification is located on the following webpage:

```
http://oss.sgi.com/projects/ogl-sample/registry/EXT/texture_compression_s3tc.txt
```

Constraints on S3TC Texture Formats

Due to the definition of the formats, the following constraints on specifying texture images and subimages in the S3TC formats:

- S3TC formats support only 2D images without borders. The function **glCompressedTexImage2DARB()** will generate a `GL_INVALID_OPERATION` error if the parameter `border` is nonzero.
- S3TC formats are block-encoded in 4x4 texel blocks and can be easily edited along block boundaries. The function **glCompressedTexSubImage2D()** will generate a `GL_INVALID_OPERATION` error if any one of the following conditions occurs:
  - The value of `width` is not a multiple of four or not equal to the value of `GL_TEXTURE_WIDTH` for the mipmap level being specified.
  - The value of `height` is not a multiple of four or not equal to the value of `GL_TEXTURE_HEIGHT` for the mipmap level being specified.
  - Either the value of `xoffset` or `yoffset` is not a multiple of four.

Note that these constraints represents a relaxation of the tighter constraints on generic compressed texture formats.

EXT_texture_filter_anisotropic—The Anisotropic Texture Filtering Extension

The EXT_texture_filter_anisotropic extension supports improved texture sampling compared to the standard mipmapping technique.

Why Use Anisotropic Texturing?

Texture mapping as defined in core OpenGL assumes that the projection of the pixel filter footprint into texture space is a square (that is, *isotropic*). In practice, however, the footprint may be long and narrow (that is, *anisotropic*). Consequently, mipmap filtering severely blurs images on surfaces angled obliquely away from the viewer. For example,

in flight simulations, views of the runway during approaches are likely to be oversampled across the runway and undersampled along its length.

There are several approaches for improving texture sampling by accounting for the anisotropic nature of the pixel filter footprint into texture space. This extension provides a general mechanism for supporting such filtering schemes without specifying a particular formulation of anisotropic filtering.

The maximum degree of anisotropy to account for in texture filtering may be defined per texture object, subject to a global upper bound determined by the implementation.

Increasing the degree of anisotropy will generally improve texture filtering quality, but at the cost of reducing the texture fill rate. Rather than setting the maximum possible anisotropy, choose the smallest degree of anisotropy that will provide the desired level of image quality and performance and consider providing interactive controls to allow users to adjust the anisotropy level further at run time.

Using Anisotropic Texturing

To specify the degree of texture anisotropy, call **glTexParameterf()** with the following parameter specifications:

target	`GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D`
pname	`GL_TEXTURE_MAX_ANISOTROPY_EXT`
param	A value between `2.0` and the implementation-dependent maximum (which may be determined by calling **glGetFloatv()** with *pname* set to `GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT`)

When the specified value of `GL_TEXTURE_MAX_ANISOTROPY_EXT` is `1.0`, standard mipmap texture sampling is used as defined in core OpenGL. When the value is greater than 1.0, a texture filtering scheme that accounts for a degree of anisotropy defined by the minimum of the specified value and the value of `GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT` is used.

While the exact anisotropic filtering scheme may vary, it will satisfy the following conditions:

- Mipmap levels will only be accessed if the texture minification filter is one that requires mipmaps.

- Anisotropic texturing will only access texture mipmap levels between the values of `GL_TEXTURE_BASE_LEVEL` and `GL_TEXTURE_MAX_LEVEL`.
- The values specified for `GL_TEXTURE_MAX_LOD` and `GL_TEXTURE_MIN_LOD` will be honored if the anisotropic scheme allows such.
- When the value of `GL_TEXTURE_MAX_ANISOTROPY_EXT` is *N*, the anisotropic filter will try to sample *N* texels within the texture footprint of the fragment being textured, where mipmapping would only sample one texel. For example, if *N* is `2.0` and the `GL_LINEAR_MIPMAP_LINEAR` filter is being used, the anisotropic filter will sample 16 texels, rather than the 8 samples used by mipmapping. However, subject to the constraints of the particular anisotropic filter being used, *N* may be rounded up at sampling time.

EXT_texture_rectangle—The Rectangle Texture Extension

OpenGL texturing is normally limited to images with power-of-two dimensions and an optional one-texel border. The EXT_texture_rectangle extension adds a new texture target that supports 2D textures without requiring power-of-two dimensions and accesses the texture by texel coordinates instead of the normalized [0,1] access used for other texture targets.

Why Use Rectangle Textures?

Rectangle (non-power-of-two) textures are useful whenever working with texture images that have such dimensions. Representing such images at their natural resolutions avoids resampling artifacts and saves texture memory. Examples include (but are not limited to) video images, shadow maps, window-space texturing, and data arrays for general-purpose computation in fragment programs.

However, rectangle textures have the following additional constraints that may restrict their applicability relative to power-of-two textures:

- Mipmaps are not supported. Rectangle textures may only define a base level image, and the minification filter must be `GL_NEAREST` or `GL_LINEAR`.
- Only the clamped texture coordinate wrap modes are allowed for the s and t coordinates: `GL_CLAMP`, `GL_CLAMP_TO_EDGE`, and `GL_CLAMP_TO_BORDER`. Repeated and/or mirrored wrap modes are not supported.
- Texture border images are not supported (*border* must be zero).

Using Rectangle Textures

To enable or disable rectangle texture mapping, call **glEnable()** or **glDisable()** with parameter `GL_TEXTURE_RECTANGLE_EXT`. When several types of of textures are enabled, the precedence order is the following:

1. `GL_TEXTURE_2D`
2. `GL_TEXTURE_RECTANGLE_EXT`
3. `GL_TEXTURE_3D`

This means that if both 2D and rectangle texturing are enabled, the currently bound rectangle texture will be used. If both 3D and rectangle texturing are enabled, the currently bound 3D texture will be used.

To define a rectangle texture, call **glTexImage2D()** or **glCopyTexImage2D()** with the parameter target set to `GL_TEXTURE_RECTANGLE_EXT`. The mipmap level and border size must both be zero. The dimensions of rectangle textures are not restricted to powers of two but are limited to the implementation-dependent maximum rectangle texture size, which can be queried by calling **glGetIntegerv()** with parameter `GL_MAX_RECTANGLE_TEXTURE_SIZE_EXT`.

Using a texture target of `GL_TEXTURE_RECTANGLE_EXT`, you can perform all other operations on rectangle textures (binding texture objects, specifying subimages, querying texture images, setting texture and texture level parameters). Using target `GL_PROXY_TEXTURE_RECTANGLE_EXT`, you can perform proxy texture queries on rectangle textures. The currently bound rectangle texture object may be queried by calling **glGetIntegerv()** with the parameter `GL_TEXTURE_BINDING_RECTANGLE_EXT`.

When rendering with a rectangle texture, texture coordinates are interpreted differently. Rather than clamping to the range [0,1], the s coordinate is clamped to the range [0,w] and the t coordinate is clamped to the range [0,h], where w and h are respectively the width and height of the rectangle texture. After clamping, you access texels directly using the clamped texture coordinates as indices into the rectangle texture, instead of first scaling them by the dimensions of the texture image as you do for normal power-of-two 2D textures.

EXT_texture3D—The 3D Texture Extension

Note: This extension was promoted to a standard part of OpenGL 1.2. For new applications, use the equivalent OpenGL 1.2 interface (for example, with the EXT suffix removed), unless they must run on InfiniteReality systems.

The 3D texture extension, EXT_texture3D, defines 3D 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 8-1.

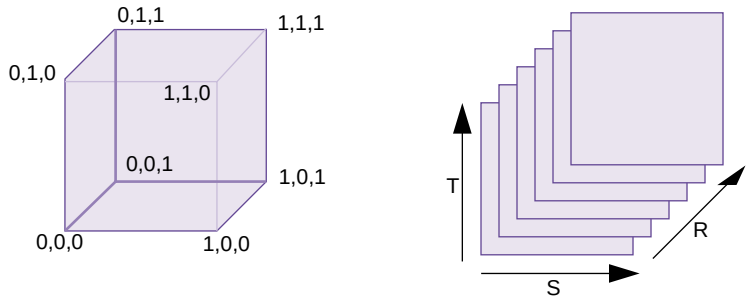


Figure 8-1 3D Texture

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

Why Use the 3D Texture Extension?

3D textures are useful for the following:

- 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 8-2**.

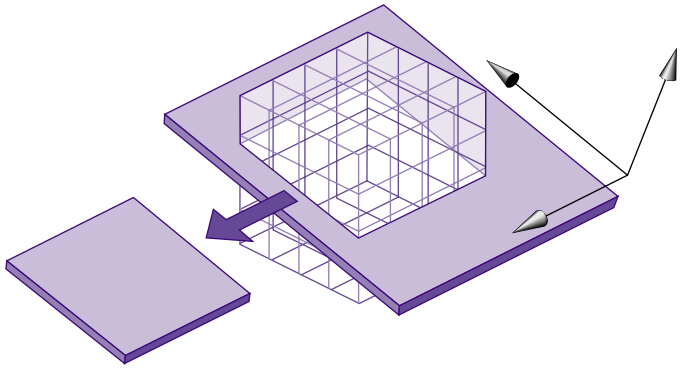


Figure 8-2 Extracting a Planar Texture From a 3D Texture Volume

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 format:

```
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 the number of “slices” in the texture.

The extension provides the following additional features:

- Pixel storage modes

  The extension extends the pixel storage modes by adding eight 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`.

  The eight state variables default to zero.

- Texture wrap modes

  The functions **glTexParameter*()** accept the additional token value `GL_TEXTURE_WRAP_R_EXT`. The value `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 described 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*`)`

- Replacing texture images

  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 8-1 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 window
 * setup, go here.
 */
```

New Functions

The EXT_texture3D extension introduces the following functions:

- **glTexImage3DEXT()**
- **glTexSubImage3DEXT()**
- **glCopyTexImage3DEXT()**

SGI_texture_color_table—The Texture Color Table Extension

Note: This extension is not supported on Onyx4 and Silicon Graphics Prism systems. Applications can achieve similar functionality by writing fragment programs using 1D textures as lookup tables for the texel values returned by sampling an image texture.

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?

The following 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" on page 277. Use `GL_TEXTURE_COLOR_TABLE_SGI` as the value for the *target* parameter of the various commands. Note 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 determine 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 Table 8-6.

Table 8-6 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 Fuel, Infinite Performance, and InfiniteReality systems. For a detailed discussion of machine-dependent issues, see the `glColorTableParameterSGI` man page. This section summarizes the most noticeable restrictions.

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 valid. InfiniteReality systems support the combinations shown in Table 8-7.

Table 8-7 Texture and Texture Color Tables on InfiniteReality Systems

TCT size	TCT Format	Texture
>=1024	Any	Any

Table 8-7 Texture and Texture Color Tables on InfiniteReality Systems **(continued)**

TCT size	TCT Format	Texture
2048	L, I, LA	L, I, LA
4096	I, L	I, L

SGIS_detail_texture—The Detail Texture Extension

Note: This extension is not supported on Onyx4 and Silicon Graphics Prism systems. Applications can achieve similar functionality using fragment programs.

This section describes the detail texture extension, SGIS_detail_texture, which like the sharpen texture extension (see "SGIS_sharpen_texture—The Sharpen Texture Extension" on page 180) 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" on page 180).

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 describes 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 the following:

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

   The internal format of the detail texture and the base texture must match exactly.

7. Set the GL_DETAIL_TEXTURE_LEVEL_SGIS parameter to specify the level at which the detail texture resides. In the road example, the detail texture is level -2 (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 8-8. For example, to apply the detail texture to both alpha and color components, use the following:

   ```
   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 8-8 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 8-9, depending on the detail texture mode.

Table 8-9 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 8-3 shows the default linear interpolation curve and a nonlinear interpolation curve that minimizes aliasing when interpolating between two LODs.

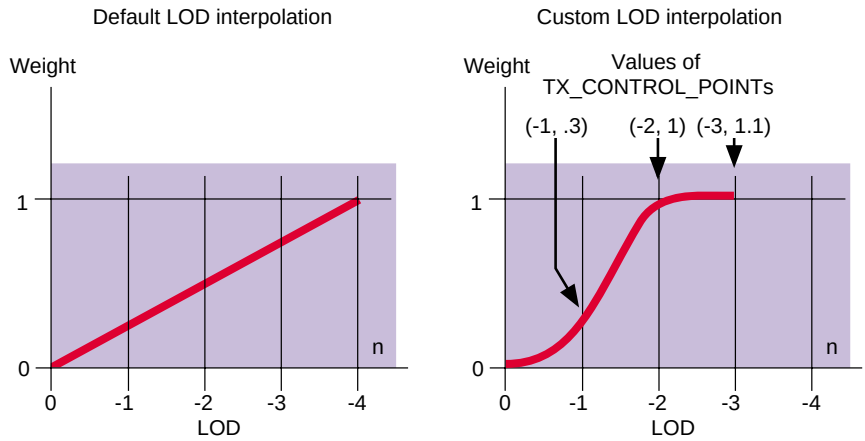


Figure 8-3 LOD Interpolation Curves

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

Use **glDetailTexFuncSGIS()** to specify control points for shaping the LOD interpolation function. Each control point contains a pair of values; the first value specifies the LOD, and the second value specifies the weight for that magnification level. 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 8-3):

```
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 8-2 is 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 8-2 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

The SGIS_detail_texture extension introduces the following functions:

- **glDetailTexFuncSGIS()**
- **glGetDetailTexFuncSGIS()**

SGIS_filter4_parameters—The Filter4 Parameters Extension

Note: This extension is part of GLU and is only supported on InfiniteReality systems. Applications can achieve higher-quality texture filtering on Onyx4 and Silicon Graphics Prism systems using anisotropic texture filtering.

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” on page 187).

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

  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

  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:

Argument	**Value**
target	`GL_TEXTURE_1D` or `GL_TEXTURE_2D`
filterype	`GLU_LAGRANGIAN_SGI` or `GLU_MITCHELL_NETRAVALI_SGI`
params	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	A power of two plus one and must be less than or equal to 1025.
weights	Pointing to 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" on page 187 for more information.

SGIS_point_line_texgen—The Point or Line Texture Generation Extension

Note: This extension is only supported on InfiniteReality systems. Applications can achieve similar functionality on Onyx4 and Silicon Graphics Prism systems by writing fragment programs.

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*, describes 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 one matching 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

Note: This extension is not supported on Onyx4 and Silicon Graphics Prism systems. Applications can achieve similar functionality using fragment programs.

This section describes the sharpen texture extension, SGIS_sharpen_texture. This extension and the detail texture extension (see "SGIS_detail_texture—The Detail Texture Extension" on page 170) 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. The following are two examples:

- 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 8-10.

Table 8-10 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 the following:

```
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, as expressed in the following formula:

```
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 described 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 8-4 shows LOD extrapolation curves as a function of magnification factors. The curve on the left is the default linear extrapolation, where weight(n) = $-n/4$. The curve on the right is a nonlinear extrapolation, where the LOD extrapolation function is modified to control the amount of sharpening so that less sharpening is applied as the magnification factor increases. The function is defined for n less than or equal to 0.

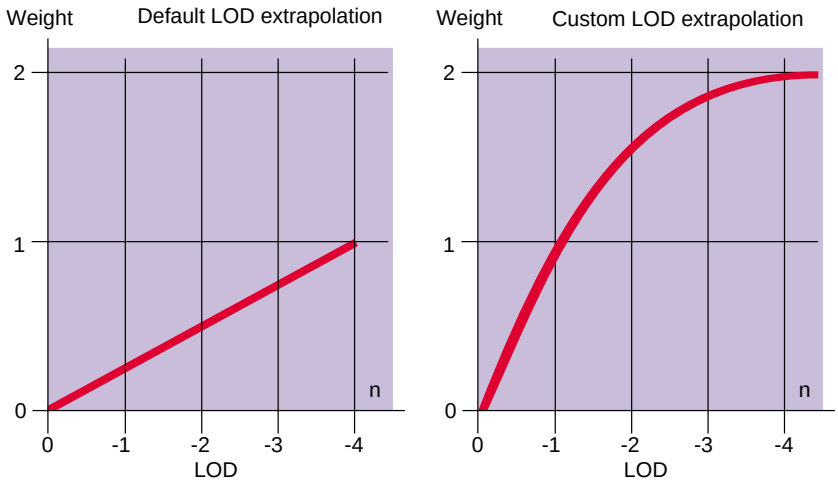


Figure 8-4 LOD Extrapolation Curves

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

For example, to gradually ease the sharpening effect, use a nonlinear LOD extrapolation curve—as shown on the right in Figure 8-4—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 8-3 illustrates 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 8-3 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

The SGIS_sharpen_texture extension introduces the following functions:

- **glSharpenTexFuncSGIS()**
- **glGetSharpenTexFuncSGIS()**

SGIS_texture_edge/border_clamp—Texture Clamp Extensions

Note: These extensions were promoted to standard parts of OpenGL 1.2 and OpenGL 1.3, respectively. Use the equivalent OpenGL interfaces (for example, with the SGIS suffixes removed) with new applications, unless they must run on InfiniteReality or InfinitePerformance systems.

This section first provides some background information on texture clamping. It then identifies reasons for using the following texture clamping extensions and explains how to use them:

- 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. This action takes 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 1D, 2D, and 3D 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()** with the following specifications:

Parameter	Value
target	`GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D_EXT`
pname	`GL_TEXTURE_WRAP_S`, `GL_TEXTURE_WRAP_T`, or `GL_TEXTURE_WRAP_R_EXT`
param	`GL_CLAMP_TO_EDGE_SGIS` for edge clamping `GL_CLAMP_TO_BORDER_SGIS` for border clamping

SGIS_texture_filter4—The Texture Filter4 Extensions

Note: This extension is only supported on InfiniteReality systems. Applications can achieve higher quality texture filtering on Onyx4 and Silicon Graphics Prism systems using anisotropic texture filtering.

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*, describes 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" on page 188). 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` man page) with the following specifications:

Parameter	Value
target	`GL_TEXTURE_1D` or `GL_TEXTURE_2D`
filter	`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 expressed as follows:

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

Each element *weights[i]* is the value of the following expression:

`f((2*i)/(`*n*`-1)), 0<=i<=`*n*`-1`

OpenGL stores and uses the filter function as a set of samples, expressed as follows:

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

The *Size* variable 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" on page 177 provides interpolation coefficients just as they are required for GL_FILTER4_SGIS filtering.

The variable *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 the following specifications:

Parameter	Value
pname	`GL_TEXTURE_MIN_FILTER` or `GL_TEXTURE_MAG_FILTER`
param or *params*	`FILTER4_SGIS`
param(s)	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

The SGIS_texture_filter4 extension introduces the following functions:

- **glTexFilterFuncSGIS()**
- **glGetTexFilterFuncSGIS()**

SGIS_texture_lod—The Texture LOD Extension

Note: This extension was promoted to a standard part of OpenGL 1.2. Use the equivalent OpenGL 1.2 interface (for example, with the SGIS suffix removed) with new applications, unless they must run on InfiniteReality systems.

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 described in this section, you should be familiar with the issues described 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*()** with the following specifications:

Parameter	**Value**
target	`GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D_EXT`
pname	`GL_TEXTURE_MIN_LOD_SGIS` or `GL_TEXTURE_MAX_LOD_SGIS`
param (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 while 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 use the following specifications:

Parameter	Value
target	`GL_TEXTURE_1D`, `GL_TEXTURE_2D`, or `GL_TEXTURE_3D_EXT`
pname	`GL_TEXTURE_BASE_LEVEL_SGIS` to specify a base level or `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

Note: This extension is only supported on InfiniteReality systems. Applications requiring efficient use of texture memory on Onyx4 and Silicon Graphics Prism systems should use the OpenGL 1.3 texture compression interface together with the compressed texture format defined by the EXT_texture_compression_s3tc extension. Alternatively, these systems may support automatic texture compression on a per-application basis by setting environment variables; see the platform release notes for more details.

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

- Four 4-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) will not 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 as follows:

```
glTexImage[n]D[EXT] ( /* Definition */
     internalFormat =
         GL_DUAL_{ ALPHA, LUMINANCE, INTENSITY * }{4, 8, 12, 16 }_SGIS
         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 8-11.

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_SGIS
        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 standard RGBA to the new internal formats is as follows:

Table 8-11 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 goes to the second group. BLUE component goes to the third group. ALPHA component goes 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

Note: This extension is only supported on InfiniteReality systems. However, OpenGL Performer implements an emulation of clipmapping; therefore, applications using OpenGL Performer will be able to use extremely large textures even on systems not supporting 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 OpenGL 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" on page 194 explains the basic assumptions behind clipmaps.
- "Using Clipmaps From OpenGL" on page 197 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" on page 200 explains in more detail some of the concepts behind the steps in clipmap creation.
- "Virtual Clipmaps" on page 203 describes how to work with a virtualized clipmap, which is the appropriate solution if some levels of the clipmap do not 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 8-5). A mipmap contains a range of levels, each four times the size of the previous one. Each level (size) determines whether clipping occurs as follows:

- 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.

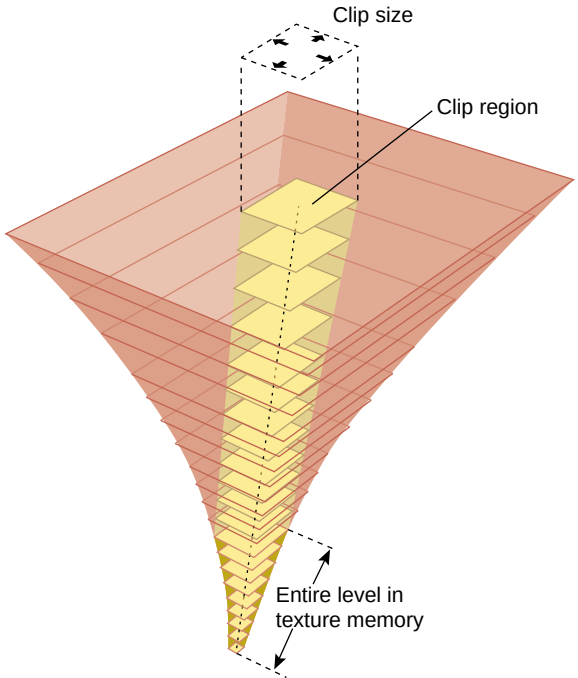


Figure 8-5 Clipmap Component Diagram

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 the following 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. The following 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" on page 202).

Using Clipmaps From OpenGL

To use clipmaps, an application has to take care of the following two distinct tasks, described 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. As shown in the following, 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 establish 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" on page 201 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" on page 200 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" on page 189) to specify the minimum and maximum LOD. See "Virtual Clipmaps" on page 203.

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 scene, the center of the clipmap usually changes with each frame. Applications, therefore, must update the clipmap stack with each frame by following these steps:

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

   See "Moving the Clip Center" on page 200 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" on page 202 discusses this in more detail.

4. Set the parameters forthe center and the offset for the next move.

Clipmap Background Information

The following sections provide background information for the steps in "Using Clipmaps From OpenGL" on page 197.

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 8-6. Applications can update the clipmaps to the new center using toroidal loading (see "Toroidal Loading" on page 202).

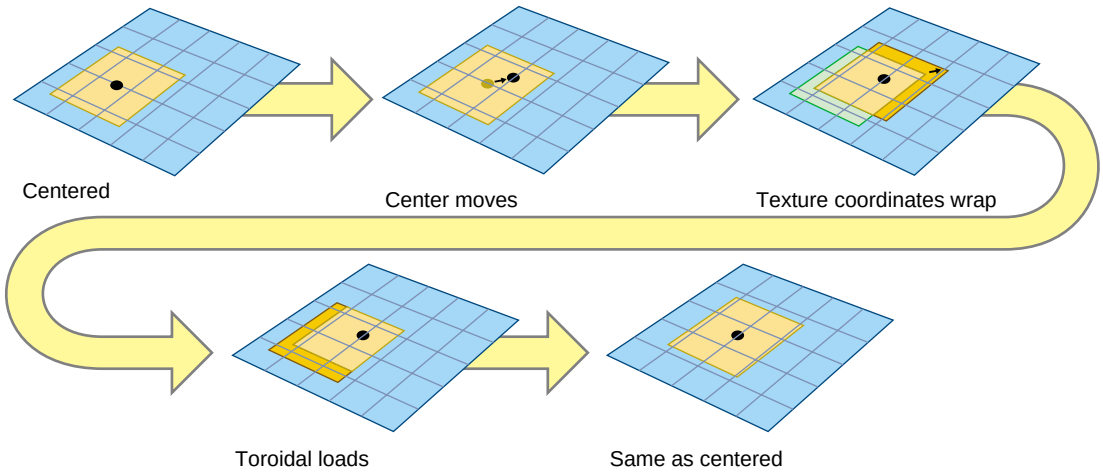


Figure 8-6 Moving the Clip Center

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.

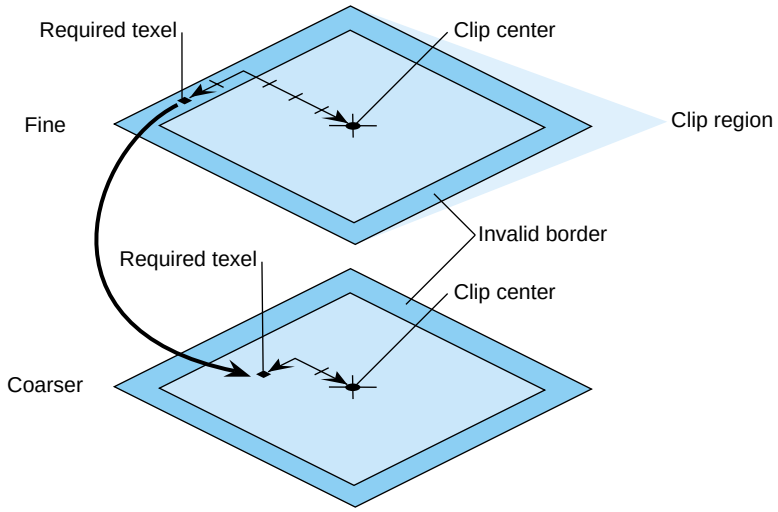


Figure 8-7 Invalid Border

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 8-6).

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

- 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 using **glTexSubImage2D()**, toroidal offset values must be added to the texture addresses of each level. The offset is specified by the

application (see “Setting Up the Clipmap Stack” on page 197). 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. Using the relative position of the viewer and the terrain to calculate the values, the application typically divides the clipmapped terrain into pieces and sets the values as each piece is traversed.

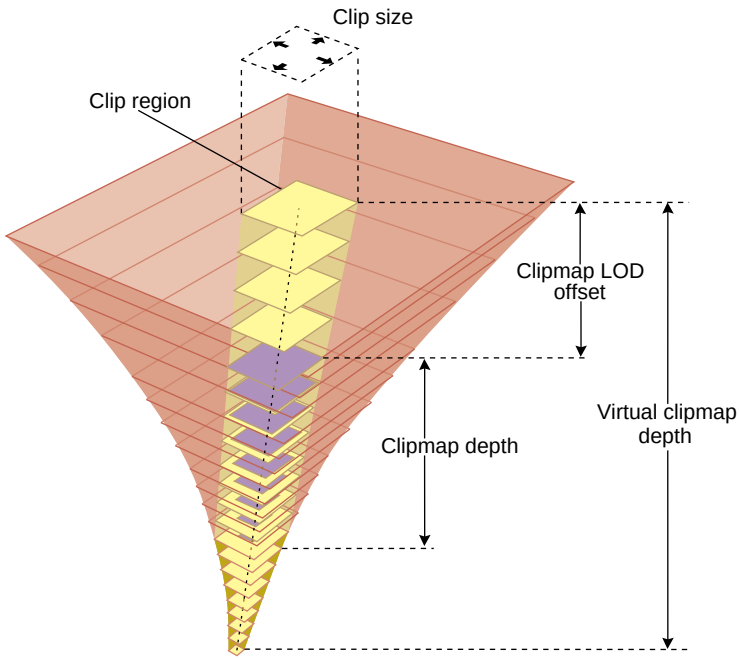


Figure 8-8 Virtual Clipmap

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 8-8 illustrates 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 the following specifications:

Parameter	**Value**
target	GL_TEXTURE_2D
pname	GL_TEXTURE_CLIPMAP_VIRTUAL_DEPTH_SGIX
params	(*D*,*N*+1,*V*+1) The value *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

Note: This extension is not supported on Onyx4 and Silicon Graphics Prism systems. Applications can achieve similar functionality by using the OpenGL 1.3 ADD texture environment mode, although the constant color scale and bias provided by this extension are not supported by base OpenGL 1.3. Alternatively, the OpenGL 1.3 texture combiner interface can be set up to match the effects of this 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 multipass 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 supports the following four texture environment functions:

- `GL_DECAL`
- `GL_REPLACE`
- `GL_MODULATE`
- `GL_BLEND`

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

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

Variable	Value
Cf	Fragment color
Cc	Constant color set by calling **glTexEnv()** with p*name* 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 described in the section "Texture Functions" on page 354 of the *OpenGL Programming Guide, Second Edition*.

SGIX_texture_lod_bias—The Texture LOD Bias Extension

Note: This extension is not supported on Onyx4 and Silicon Graphics Prism systems. Applications can achieve similar functionality by using the OpenGL 1.4 `TEXTURE_LOD_BIAS` parameter, although the numerical meaning of the bias is not identical between this extension and the OpenGL 1.4 feature because the bias is added at different stages of computing the level of detail.

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 (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 8-9 shows 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 8-10, this effect is exaggerated by a positive LOD bias. Figure 8-11 shows how the markers become visible with a negative LOD bias.

Figure 8-9 Original Image

Figure 8-10 Image With Positive LOD Bias

Figure 8-11 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 the following:

- 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 need not to be scaled down as much.
- Textures that do not have power-of-two dimensions and, therefore, they 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 the following specifications:

Parameter	Value
target	`TEXTURE_1D`, `TEXTURE_2D`, or TEXTURE_3D_EXT
pname	`GL_TEXTURE_LOD_BIAS_S_SGIX`, `GL_TEXTURE_LOD_BIAS_T_SGIX`, or `GL_TEXTURE_LOD_BIAS_R_SGIX`
param (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

Note: This extension is not supported on Onyx4 and Silicon Graphics Prism systems. Applications can achieve similar functionality by setting up the OpenGL 1.3 texture combiner interface to match the effects of this extension or by using fragment programs.

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 performing histogram normalization or gamut expansion, for example. 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 the following specifications:

Parameter	**Value**
pname	`GL_POST_TEXTURE_FILTER_BIAS_SGIX` or `GL_POST_TEXTURE_FILTER_SCALE_SGIX`
params	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.

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. The following extenstions are described in this chapter:

ATI_draw_buffers—The Multiple Draw Buffers Extension

The ATI_draw_buffers extension allows fragment programs to generate multiple output colors, and provides a mechanism for directing those outputs to multiple color buffers.

Why Use Multiple Draw Buffers?

Multiple draw buffers are typically useful when generating an image and auxiliary data, multiple versions of an image, or multiple computed results from a fragment program being used for general-purpose computation.

Using Multiple Draw Buffers

Normally, a fragment program will generate a single output color, `result.color`, which is written to the color buffer defined by **glDrawBuffer()**. When a fragment program specifies the option "`ATI_draw_buffers`", an implementation-dependent number of output colors, named `result.color[`*n*`]`, may be generated, where *n* ranges from 0 up to the number of draw buffers minus one. The number of draw buffers supported is implementation-dependent, and may be queried by calling **glGetIntegerv()** with the parameter `GL_MAX_DRAW_BUFFERS_ATI`. Typically, at least four draw buffers are supported by this extension.

To define the color buffers to which multiple output colors are written, make the following call:

```
void glDrawBuffersATI(GLsizei n, const GLenum *bufs);
```

The parameter *n* specifies the number of buffers in *bufs* and *bufs* is a pointer to an array of symbolic constants specifying the buffer to which each output color is written. The constants may be one of the following:

- `GL_NONE`
- `GL_FRONT_LEFT`
- `GL_FRONT_RIGHT`
- `GL_BACK_LEFT`
- `GL_BACK_RIGHT`

- GL_AUX0 through GL_AUX*n*, where *n* + 1 is the number of available auxiliary buffers.

The draw buffers being defined correspond in order to the respective output colors. The draw buffer for output colors beyond *n* is set to GL_NONE.

The constants GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, and GL_FRONT_AND_BACK, which may refer to multiple color buffers, are not valid elements of *bufs*, and their use will generate a GL_INVALID_OPERATION error.

If the "ATI_draw_buffers" fragment program option is not used by a fragment program or if fixed-function fragment processing is being used, then **glDrawBuffersATI()** specifies a set of draw buffers for the writing of the output color 0 or the output color from fixed-function fragment processing.

The draw buffer corresponding to output color *i* may be queried by calling **glGetIntegerv()** with the parameter GL_DRAW_BUFFER*i*_ATI.

New Function

The ATI_draw_buffers extension introduces the function **glDrawBuffersATI()**.

ATI_separate_stencil—The Separate Stencil Extension

The ATI_separate_stencil extension provides the ability to modify the stencil buffer based on the orientation of the primitive that generated a fragment.

Why Use the Separate Stencil Extension?

When performing stencil buffer computations which differ for fragments generated by front-facing and back-facing primitives, applications typically must render geometry twice. They use face culling to discard front-facing primitives with one pass and back-facing primitives on the second and change stencil buffer settings prior to each pass. A common example is stencil shadow volumes, where the stencil buffer is to be incremented for front-facing fragments and decremented for back-facing fragments.

By using independent stencil tests and operations depending on fragment orientation, such computations can be performed in a single pass, which may significantly increase performance for geometry-limited applications.

Using the Separate Stencil Extension

To set the stencil *function* separately for front-facing and back-facing fragments, make the following call:

```
void glStencilFuncSeparateATI(GLenum frontfunc, GLenum backfunc, GLint ref,
                              GLuint mask);
```

The parameters *frontfunc* and *backfunc* respectively specify the stencil test function used for front-facing and back-facing fragments. The values accepted are the same as for **glStencilFunc()**, and the initial value of each function is `GL_ALWAYS`.

The parmater *ref* specifies the reference value used for both front-facing and back-facing fragments. It is clamped to the range [0, **pow(**2,*s***)** – 1], where *s* is the number of bits in the stencil buffer.

The *s* least significant bits of the mask value are bitwise ANDed with ref and then with the stored stencil value, and the resulting masked value is used in the comparison controlled by {\em `func`}.

To set the stencil operation separately for front-facing and back-facing fragments, make the following call:

```
void glStencilOpSeparateATI(GLenum face, GLenum fail, GLenum zfail,
                            GLenum   zpass);
```

The parameter face specifies the orientation for the stencil operation and must be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK` to set both stencil operations to the same values.

The parameters *fail*, *zfail*, and *zpass* respectively specify the operations to perform when the stencil test fails, stencil test passes but depth test fails, and stencil and depth tests both pass. The values accepted are the same as for **glStencilOp()**.

Use the core OpenGL tokens to query for the front-facing stencil state. To query for the back-facing stencil state, call **glGetIntegerv()** with the following tokens:

- `GL_STENCIL_BACK_FUNC_ATI`

- GL_STENCIL_BACK_FAIL_ATI
- GL_STENCIL_BACK_PASS_DEPTH_FAIL_ATI
- GL_STENCIL_BACK_PASS_DEPTH_PASS_ATI

New Functions

The ATI_separate_stencil extension introduces the following functions:

- **glStencilFuncSeparateATI()**
- **glStencilOpSeparateATI()**

NV_point_sprite—The Point Sprite Extension

The NV_point_sprite extension supports application of texture maps to point primitives instead of using a single texture coordinate for all fragments generated by the point. Note that NV_point_sprite is not related to the SGIX_sprite extension described in section "SGIX_sprite—The Sprite Extension" on page 250.

Why Use Point Sprites?

When rendering effects such as particle systems, applications often want to draw a small texture (such as a shaded sphere) to represent each particle rather than the set of uniformly shaded fragments normally generated by a GL_POINTS primitive. This can easily be done by rendering a GL_QUADS primitive for each point but at the cost of quadrupling the amount of geometry transferred to the graphics pipeline for each point and of performing additional work to compute the location of each vertex of the quad. Since particle systems typically involves thousands or tens of thousands of particles, this can translate to a large geometry load.

Point sprites allow producing these effects using point primitives instead of quads. Each texture unit can be modified to replace the single S and T texture coordinate for each fragment generated by a point with S and T point sprite coordinates, which are interpolated across the fragments generated by a point. Finally, a global parameter controls the R texture coordinate for point sprites to allow applications to animate slices of a single 3D texture during the lifetime of a point. For example, it allows an application to represent a particle that glows and then fades.

Using Point Sprites

Point sprites are enabled by calling **glEnable(**`GL_POINT_SPRITE_NV`**)**. When point sprites are enabled, the state of point antialiasing is ignored so that fragments are generated for the entire viewport square occupied by the point instead of just fragments in a circle filling that viewport square.

When point sprites are enabled, each texture unit may independently determine whether or not the single point texture coordinate is replaced by point sprite texture coordinates by making the following call:

`glTexEnvi(GL_POINT_SPRITE_NV, GL_COORD_REPLACE_NV,` *flag*`);`

The active texture unit will generate point sprite coordinates if *flag* is `GL_TRUE` or will use the point texture coordinate if flag is `GL_FALSE`.

The point sprite texture coordinates generated for fragments by a point will be the following:

`s = 1/2 + ((`x_f `-` x_w `+ 1/2) /` *size*`)`

`t = 1/2 + ((`y_f `-` y_w `+ 1/2) /` *size*`)`

The variable items are defined as follows:

(x_f, y_f)	Specifies the window coordinates of a fragment generated by the point.
(x_w, y_w)	Specifies the floating point coordinates of the point center.
size	Specifies the screen-space point width, which depends on the current point width as well as the scaling determined by the current point parameters.

When 3D texturing is enabled, the R value generated for point sprite coordinates is determined by making the following call:

`glPointParameteriNV(GL_POINT_SPRITE_R_MODE_NV, GLint` *param*`);`

The following are possible values of *param*:

`GL_ZERO`	The R coordinate generated for all fragments will be zero. This is typically the fastest mode. `GL_ZERO` is the default.
`GL_S`	The R coordinate generated for all fragments will be taken from the S coordinate of the point before point sprite coordinates are generated.

`GL_R`	The R coordinate generated for all fragments will be taken from the R coordinate of the point before point sprite coordinates are generated.

NV_occlusion_query—The Occlusion Query Extension

The NV_occlusion_query extension provides a high-level mechanism to query the visibility of an object and returns a count of the number of pixels that pass the depth test.

Why Use Occlusion Queries?

Occlusion queries are primarily used to help applications avoid rendering objects that are completely occluded (blocked from visibility) by other objects closer to the viewer. This can result in a significantly reduced geometry load.

Typically, this test consists of the following steps:

1. Drawing large foreground objects (*occluders*) that are expected to block background objects
2. Starting the occlusion test
3. Drawing simple primitives representing the bounding box of background objects that may be occluded
4. Ending the occlusion test
5. Reading back the number of pixels of the bounding box that passed the depth test

If the number of pixels that passed the depth test is zero, then the objects represented by this bounding box are completely occluded and do not need to be drawn. Otherwise, at least some of the objects within the bounding box may be visible and can either be drawn or finer-detailed occlusion queries can be performed on smaller components of the objects. In addition, if the number of pixels is small relative to the size of the bounding box, it may be possible to represent the objects with lower-detailed models.

Some other possible uses for occlusion queries include *depth peeling* techniques like as order-independent transparency, where an application can stop rendering when further layers will be invisible, and as a replacement for **glReadPixels()** when performing operations like reading the depth buffer to determine fractional visibility of a light source for lens flare or halo effects.

Use occlusion queries with care, however. Naive use of a query may stall the graphics pipeline and CPU while waiting for query results. To avoid this problem, NV_occlusion_query supports a simple test for the availability of query results. If the query results are not available, the application can do other drawing or compute tasks while waiting for the results to become available.

In addition, the expense of rendering bounding boxes for an occlusion test, while typically small compared to the expense of rendering the objects themselves, can become significant if done too finely (for example, rendering bounding boxes for small objects) or if done frequently when it is unlikely that the bounding boxes will actually be occluded.

Using the NV_occlusion_query Extension

Occlusion queries depend on occlusion query objects. As shown in the following code, these objects are represented by object names (of type `GLuint`), which are managed in exactly the same fashion as texture and display list names—that is, with routines for allocating unused query names, deleting query names, and testing if a name is a valid occlusion query:

```
void glGenOcclusionQueriesNV(GLsizei n, GLuint *ids);
void glDeleteOcclusionQueriesNV(GLsizei n, const GLuint *ids);
GLboolean glIsOcclusionQueryNV(GLuint id);
```

Occlusion query objects contain a pixel counter, which is initially set to zero. The size (in bits) of this counter is the same for all queries and may be determined by calling **glGetIntegerv()** with parameter `GL_PIXEL_COUNTER_BITS_NV`. An occlusion query counter is guaranteed to contain at least 24 bits, supporting pixels counts of at least 16777215, but it may be larger.

To perform occlusion queries, first acquire an unused query name using **glGenOcclusionQueriesNV()**. Begin the query by making the following call:

```
void glBeginOcclusionQueryNV(GLuint id);
```

The parameter *id* specifies the name of the query to be created. Then render the geometry to be queried. Whenever a fragment being rendered passes the depth test while an occlusion query is being performed, the pixel counter is incremented by one. In a multisampling situation, the pixel counter is incremented once for each sample whose coverage bit in the fragment is set.

Typically, when rendering bounding boxes for an occlusion test, the color and depth masks are set to GL_FALSE so that the bounding boxes themselves are not drawn to the framebuffer.

To end an occlusion query, make the following call:

```
void glEndOcclusionQueryNV(void);
```

To retrieve the count of pixels that passed the occlusion query, make the following call:

```
void glGetOcclusionQueryuivNV(GLuint id, GLenum pname, GLuint *params);
```

Set the parameter *pname* to GL_PIXEL_COUNT_NV. The count is returned in the variable pointed to by *params*. However, as noted earlier, calling **glGetOcclusionQueryNV()** immediately after ending a query may cause the graphics pipeline to stall.

To avoid stalling, first determine whether the query count is available by calling **glGetOcclusionQueryuiNV()** with a *pname* of GL_PIXEL_COUNT_AVAILABLE_NV. If the value returned in params is GL_TRUE, then the count is available, and a query of GL_PIXEL_COUNT_NV may be performed without stalling. Otherwise, the application may perform additional work unrelated to the occlusion query and test periodically for the result. Note that the first call to determine GL_PIXEL_COUNT_AVAILABLE_NV for a query should be preceded by **glFlush()** to ensure that the **glEndOcclusionQueryNV()** operation for that query has reached the graphics pipeline. Otherwise, it is possible to spin indefinitely on the query.

Example 9-1 shows a simple use of NV_occlusion_query.

Example 9-1 NV_occlusion_query Example

```
GLuint occlusionQuery[numQuery]; /* names for each query to perform */

glGenOcclusionQueriesNV(numQuery, occlusionQuery);

/* Prior to this point, first render the foreground occluders */
/* Disable color and depth mask writes while rendering bounding boxes
*/
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
/* Also disable texturing, fragment shaders, and any other
 * unneccessary functionality, since nothing will actually be
 * written to the screen.
 */
```

```
/* Now loop over numQuery objects, performing an occlusion query for
each */
for (i = 0; i < numQuery; i++) {
    glBeginOcclusionQueryNV(occlusionQuery[i]);
    /* Render bounding box for object i */
    glEndOcclusionQueryNV();
}

/* Enable color and depth mask writes, and any other state disabled
 * above prior to the occlusion queries
 */
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);

/* If possible, perform other computations or rendering at this
 * point, while waiting for occlusion results to become available.
 */

/* Now obtain pixel counts for each query, and draw objects based
 * on those counts.
 */
for (i = 0; i < numQuery; i++) {
    GLuint pixelCount;

    glGetOcclusionQueryuivNV(occlusionQuery[i], GL_PIXEL_COUNT_NV,
                             &pixelCount);

    if (pixelCount > 0) {
        /* Render geometry for object i here */
    }
```

New Functions

The NV_occlusion_query extension introduces the following functions:

- **glGenOcclusionQueriesNV()**
- **glDeleteOcclusionQueriesNV()**
- **glIsOcclusionQueryNV()**
- **glBeginOcclusionQueryNV()**
- **glEndOcclusionQueryNV()**

- **glGetOcclusionQueryuivNV()**

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` specified.

Blending is described in the first section of Chapter 7, "Blending, Antialiasing, Fog, and Polygon Offset" of the *OpenGL Programming Guide*. The section also lists a number of sample uses of blending.

This section explains how to use extensions that support color blending for images and rendered geometry in a variety of ways:

- "Constant Color Blending Extension"
- "Minmax Blending Extension"
- "Blend Subtract Extension"

Note: These three extensions were promoted to a standard part of OpenGL 1.2. Use the equivalent OpenGL 1.2 interfaces (for example, with the EXT suffixes removed) with new applications, unless they must run on InfiniteReality, InfinitePerformance, or Fuel systems. The extensions are supported on all current Silicon Graphics systems.

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()**, whose format follows, 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 9-1 as source or destination factor, or both.

Table 9-1 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_EXT);
for (alpha = 0.0; alpha <= 1.0; alpha += 1.0/16.0) {
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, image0);
    glEnable(GL_BLEND);
    glBlendColorEXT(alpha, alpha, alpha, alpha);
    glDrawPixels(width, height, GL_RGB, GL_UNSIGNED_BYTE, image1);
    glDisable(GL_BLEND);
```

```
        glXSwapBuffers(display, window);
    }
```

New Functions

The EXT_blend_color extension introduces the function **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()**, whose format follows:

`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” on page 224. 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

The EXT_BLEND_MINMAX extension introduces the function **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 man 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, InfinitePerformance, and Fuel systems. Applications can achieve similar functionality on Onyx4 and Silicon Graphics Prism systems using fragment programs.

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.

Replacing any previous specification that may have existed, the *n* value pairs in *points* completely specify the function. At least one control point should be specified. The maximum number of control points is implementation-dependent and may be retrieved by calling **glGetIntegerv()** with a *pname* of `GL_MAX_FOG_FUNC_POINTS_SGIS` while the number of points actually specified for the current fog function may be retrieved with a *pname* of `FOG_FUNC_POINTS_SGIS`.

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 the fog function extension can be executed well only on those platforms where the extension is supported (VPro and InfiniteReality systems).

```
#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.
 *
 * 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 default */
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 enable */
    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

The SGIS_fog_function extension introduces the function **glFogFuncSGIS()**.

SGIS_fog_offset—The Fog Offset Extension

Note: This extension is not supported on Onyx4 and Silicon Graphics Prism systems. Applications can achieve similar functionality using fragment programs.

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* value 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` value 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 eye space 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.

The Multisample Extension

There are two version of the multisample extension:

- ARB_multisample
- SGIS_multisample

Note: Functionality-wise, the ARB and SGIS versions of this extension are very similar but not identical. The SGIS version is only supported on InfiniteReality systems. The extension has been promoted to a standard ARB extension, and new applications should use the equivalent ARB interface, unless they must run on InfiniteReality systems. The ARB version of this extension is only supported on Silicon Graphics Prism systems.

SGIS_multisample differs from ARB_multisample in the following respects:

- All SGIS suffixes on function and token names are changed to ARB.
- The term `mask` is changed to `coverage` in token and function names.
- The ability to change the sample pattern between rendering passes, described in section "Accumulating Multisampled Images" on page 236, is only supported by the SGIS version of the extension.

Table 9-2 shows the overall mapping between SGIS and ARB tokens and functions.

Table 9-2 Mapping of SGIS and ARB tokens for Multisampling

SGIS_multisample Token	ARB_multisample Token
SampleMaskSGIS	SampleCoverageARB
GLX_SAMPLE_BUFFERS_SGIS	GLX_SAMPLE_BUFFERS_ARB
GLX_SAMPLES_SGIS	GLX_SAMPLES_ARB
MULTISAMPLE_SGIS	MULTISAMPLE_ARB
SAMPLE_ALPHA_TO_MASK_SGIS	SAMPLE_ALPHA_TO_COVERAGE_ARB
SAMPLE_ALPHA_TO_ONE_SGIS	SAMPLE_ALPHA_TO_ONE_ARB
SAMPLE_MASK_SGIS	SAMPLE_COVERAGE_ARB
MULTISAMPLE_BIT_EXT	MULTISAMPLE_BIT_ARB

Table 9-2 Mapping of SGIS and ARB tokens for Multisampling **(continued)**

SGIS_multisample Token	ARB_multisample Token
SAMPLE_BUFFERS_SGIS	SAMPLE_BUFFERS_ARB
SAMPLES_SGIS	SAMPLES_ARB
SAMPLE_MASK_VALUE_SGIS	SAMPLE_COVERAGE_VALUE_ARB
SAMPLE_MASK_INVERT_SGIS	SAMPLE_COVERAGE_INVERT_ARB
SamplePatternSGIS	Not supported
SAMPLE_PATTERN_SGIS	Not supported
1PASS_SGIS	Not supported
2PASS_0_SGIS	Not supported
2PASS_1_SGIS	Not supported
4PASS_0_SGIS	Not supported
4PASS_1_SGIS	Not supported
4PASS_2_SGIS	Not supported
4PASS_3_SGIS	Not supported

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 describes the following topics:

- "Introduction to Multisampling" on page 232
- "Using the Multisample Extension" on page 232 and "Using Advanced Multisampling Options" on page 233
- "How Multisampling Affects Different Primitives" on page 237

Introduction to Multisampling

Multisampling works by sampling all primitives multiple times at different locations within each pixel; in effect, multisampling collects 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 in 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" on page 237 and "Multisampled Lines" on page 237 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. The function **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, make the following call:

```
glIsEnabled(MULTISAMPLE_SGIS)
```

To turn off multisampling, make the following call:

```
glDisable(MULTISAMPLE_SGIS)
```

Using Advanced Multisampling Options

Advanced multisampling options provide additional rendering capabilities. This section describes the following features:

- 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

The following steps, illustrated in Figure 9-1, shows how the subsamples in one pixel are turned on and off.

1. The primitive is sampled at the locations defined by a sample pattern. If a sample is inside the polygon, it is turned on; otherwise, it is turned off. This produces a coverage mask.
2. The coverage mask is then ANDed with a user-defined sample mask, defined by a call to **glSampleMaskSGIS() (**see "Using a Multisample Mask to Fade Levels of Detail" on page 235).
3. You may also choose to convert the alpha value of a fragment to a mask and AND it with the coverage mask from step 2.

   Enable GL_SAMPLE_ALPHA_TO_MASK_SGIS to convert alpha to the mask. The fragment alpha value is used to generate a temporary mask, which is then ANDed with the fragment mask.

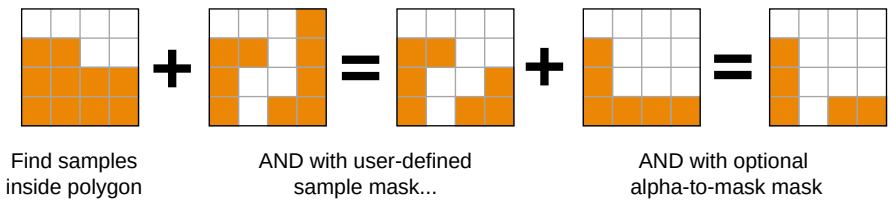


Figure 9-1 Sample Processing During Multisampling

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" on page 236 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. This results in a transparent image. This is useful because it does not require the polygons to be sorted from back to front. It is also faster because it does not 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 fading the 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 (described in the previous section), either call **glSampleMaskSGIS()** or use the alpha values of the object and call **glSampleAlphaToMaskSGIS().**

The following is the format for **glSampleMaskSGIS()**:

```
void glSampleMaskSGIS (GLclampf value, boolean invert)
```

The parameters are defined as follows:

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 and 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 describes 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 antialiased points, remember that you can turn multisampling on and off as needed to achieve multisampled polygons and antialiased points.

Note: On InfiniteReality systems, you achieve higher-quality multisampled points by setting `GL_POINT_SMOOTH_HINT` 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 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 line width. Rectangle length is exactly equal to the length of the segment. Rectangles of colinear, abutting line segments abut exactly so that no subsamples are missed or drawn twice near the shared vertex.

Just like points, lines on 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 antialiased 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.

New Functions

The SGIS_multisample extension introduces the following functions:

- **glSampleMaskSGIS()**
- **glSamplePatternSGIS()**

The Point Parameters Extension

There are two versions of the point parameters extension:

- ARB_point_parameters
- SGIS_point_parameters

Note: Functionality-wise, the ARB and SGIS versions of this extension are identical. The SGIS version is only supported on InfiniteReality systems. The extension has been promoted to a standard ARB extension, and new applications should use the equivalent ARB interface, unless they must run on InfiniteReality systems. The ARB version of this extension is only supported on Silicon Graphics Prism systems.

The following descriptions refer to the SGIS version of the extension. When using the ARB version, simply replace the `SGIS` suffix on function and token names with `ARB`, except (as noted later) for `GL_DISTANCE_ATTENUATION_SGIS`. In this case, use `GL_POINT_DISTANCE_ATTENUATION_ARB` instead.

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.

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 the following 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 with 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.

`GL_DISTANCE_ATTENUATION_SGIS`
: Defines coefficients of the distance attenuation function. In the ARB version of this extension, use the constant `GL_POINT_DISTANCE_ATTENUATION_ARB`.

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
: `GL_POINT_SIZE_MIN_SGIS`, `GL_POINT_SIZE_MAX_SGIS`, or `GL_POINT_FADE_THRESHOLD_SIZE_SGIS` (multisample points only) `GL_DISTANCE_ATTENUATION_SGIS` (In the ARB version of this extension, use `GL_POINT_DISTANCE_ATTENUATION_ARB`.)

param When *pname* is `GL_POINT_SIZE_MIN_SGIS`, `GL_POINT_SIZE_MAX_SGIS`, or `GL_POINT_FADE_THRESHOLD_SIZE_SGIS`, *param* is respectively set to the single numeric value you want to set for the minimum size, maximum size, or threshold size of the point. When *pname* is `GL_DISTANCE_ATTENUATION_SGIS`, *param* is a pointer to an array of three coefficients in order: *a*, *b*, and *c*, defining the distance attention coefficients for point size. The distance attenuation equation is described in section "Point Parameters Background Information" on page 242.

Note: If you are using the extension in multisample mode, you must use smooth points to achieve the desired improvements, as shown in the following:

```
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" on page 242 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 9-2 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, fadeSize );
        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:

```
dist_atten(d) = 1 / (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 you provide through this extension.

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

New Procedures and Functions

The SGIS_point_parameters extension introduces the following functions:

- **glPointParameterfSGIS()**
- **glPointParameterfvSGI()**

The ARB_point_parameters extension introduces the following functions:

- **glPointParameterfARB()**
- **glPointParameterfvARB()**

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 the following:

- 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

The SGIX_reference_plane extension introduces the function **glReferencePlaneSGIX()**.

The Shadow Extensions

The following are the ARB and SGIX versions of the three shadow extensions:

- ARB_depth_texture
- ARB_shadow
- ARB_shadow_ambient
- SGIX_depth_texture
- SGIX_shadow
- SGIX_shadow_ambient

Note: Functionality-wise, the ARB and SGIX versions of these extension are identical. The SGIX versions are only supported on InfiniteReality systems. The extensions have been promoted to standard ARB extensions, and new applications should use the equivalent ARB interface, unless they must run on InfiniteReality systems. The ARB versions of these extensions are only supported on Silicon Graphics Prism systems.

The following descriptions refer to the SGIX version of the extension. When using the ARB version, simply replace the `SGIX` suffix on function and token names with `ARB`, except (as noted later) for `GL_SHADOW_AMBIENT_SGIX`. In this case, use `GL_TEXTURE_COMPARE_FAIL_VALUE_ARB` instead.

This section describes three SGIX extensions that are used together to create shadows:

SGIX_depth_texture	Defines a new depth texture internal format. While this extension has other potential uses, it is currently used for shadows only.
SGIX_shadow	Defines two operations that can be performed on texture values before they are passed to the filtering subsystem.
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 describes 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 one of the following internal formats:

- `GL_DEPTH_COMPONENT`
- `GL_DEPTH_COMPONENT16_SGIX`
- `GL_DEPTH_COMPONENT24_SGIX`
- `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 does not 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);
}
```

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 9-2 and Figure 9-3 contrast rendering from the normal viewpoint and the light source viewpoint.

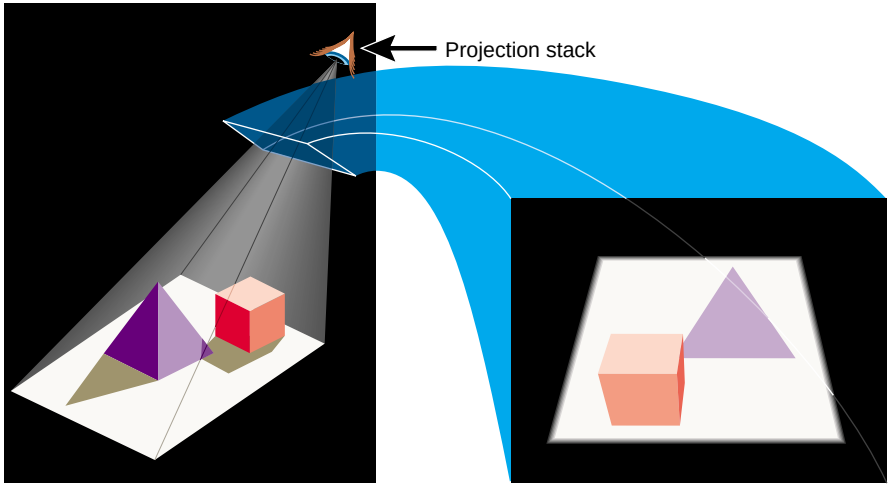


Figure 9-2 Rendering From the Light Source Point of View

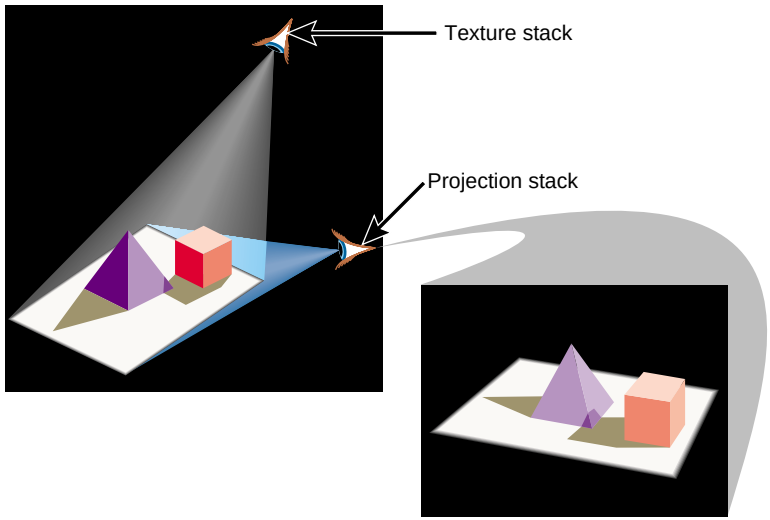


Figure 9-3 Rendering From Normal Viewpoint

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 described 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 the following parameter specifications:

pname	`GL_SHADOW_AMBIENT_SGIX` (`GL_TEXTURE_COMPARE_FAIL_VALUE` in the ARB version)
param	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 following:

- 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

Depending on the sprite mode, primitives are transformed by a rotation, as described in the following:

`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 9-4 for 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 9-5 for 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 9-6 for 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.

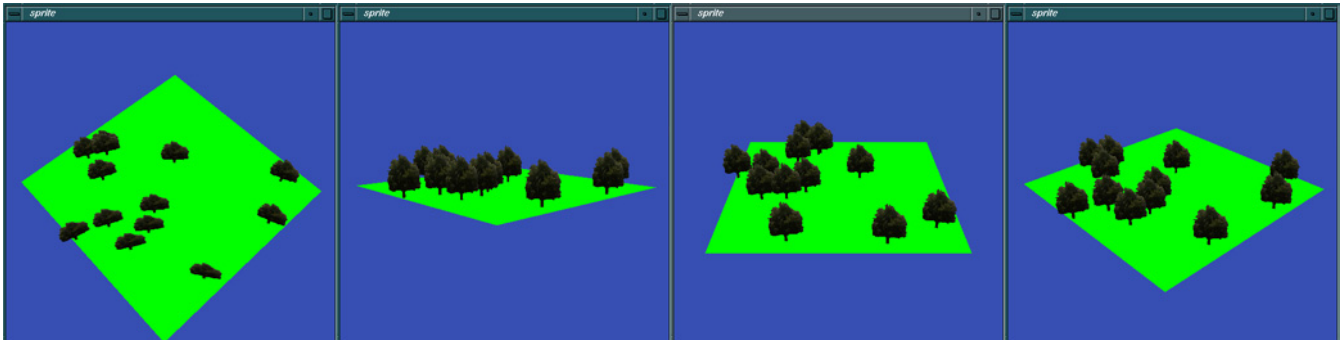


Figure 9-4 Sprites Viewed with Axial Sprite Mode

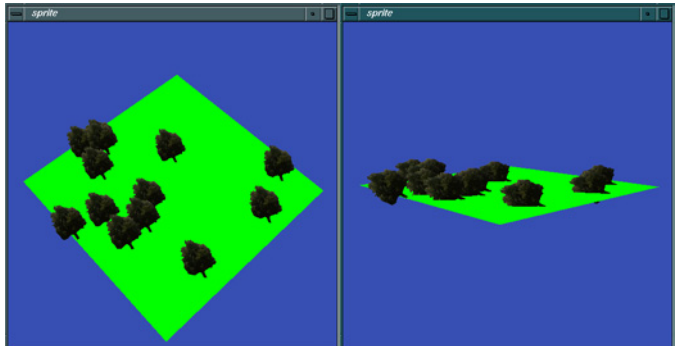


Figure 9-5 Sprites Viewed With Object Aligned Mode

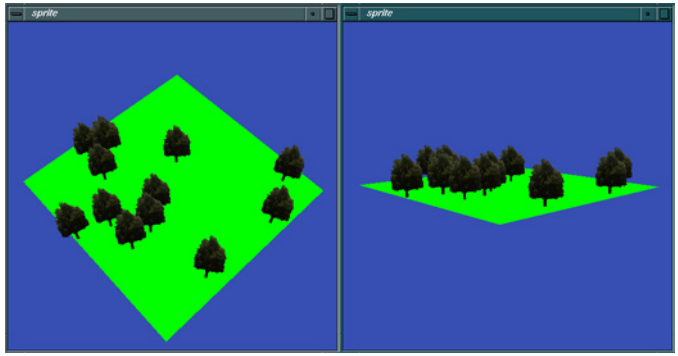


Figure 9-6 Sprites Viewed With Eye Aligned Mode

Note: The sprite extension specification describes in more detail how the sprite transformation is computed. See "Extension Specifications" on page 110 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`
   - `GL_SPRITE_EYE_ALIGNED_SGIX`
3. Specify the axis of rotation and the translation.
4. Draw the sprite geometry.
5. 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 9-3 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_ALIGNED_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_ALIGNED_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 standard frame: +Z is the up vector. -Y is the front vector, which is rotated to point towards the eye.

New Function

The SGIX_sprite extension introduces the function **glSpriteParameterSGIX().**

Imaging Extensions

This chapter describes imaging extensions. After some introductory information the imaging pipeline, the following extensions are described:

Introduction to Imaging Extensions

This section describes platform dependencies, where extensions are in the OpenGL imaging pipeline, and the functions that may be affected by one of the imaging extensions.

Platform Dependencies

Currently, the majority of the imaging extensions are only supported on Fuel, InfinitePerformance, and InfiniteReality systems. The imaging extensions supported on Onyx4 and Silicon Graphics Prism systems include only the following:

- EXT_abgr
- EXT_packed_pixels

- SGI_color_matrix

The EXT_packed_pixels extension was promoted to a standard part of OpenGL 1.2 and is available in that form.

Applications on Onyx4 and Silicon Graphics Prism systems can achieve similar functionality to the SGI_color_table and SGIX_pixel_texture extensions by writing fragment programs using one-dimensional textures as lookup tables.

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 10-1 is a high-level illustration of pixel paths.

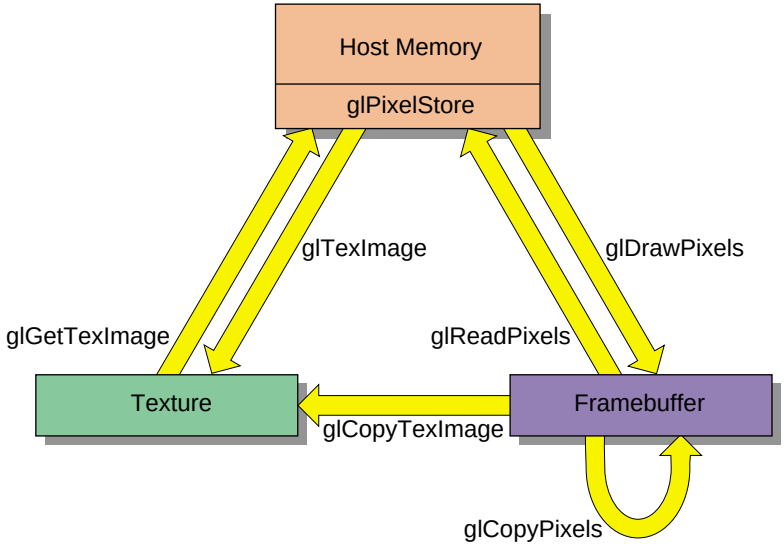


Figure 10-1 OpenGL Pixel Paths

The OpenGL pixel paths show the movement of rectangles of pixels among 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()** function and other operations in the pixel transfer pipeline apply to all paths among 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 10-2. The pixel transfer pipeline is not applied to any of these paths.

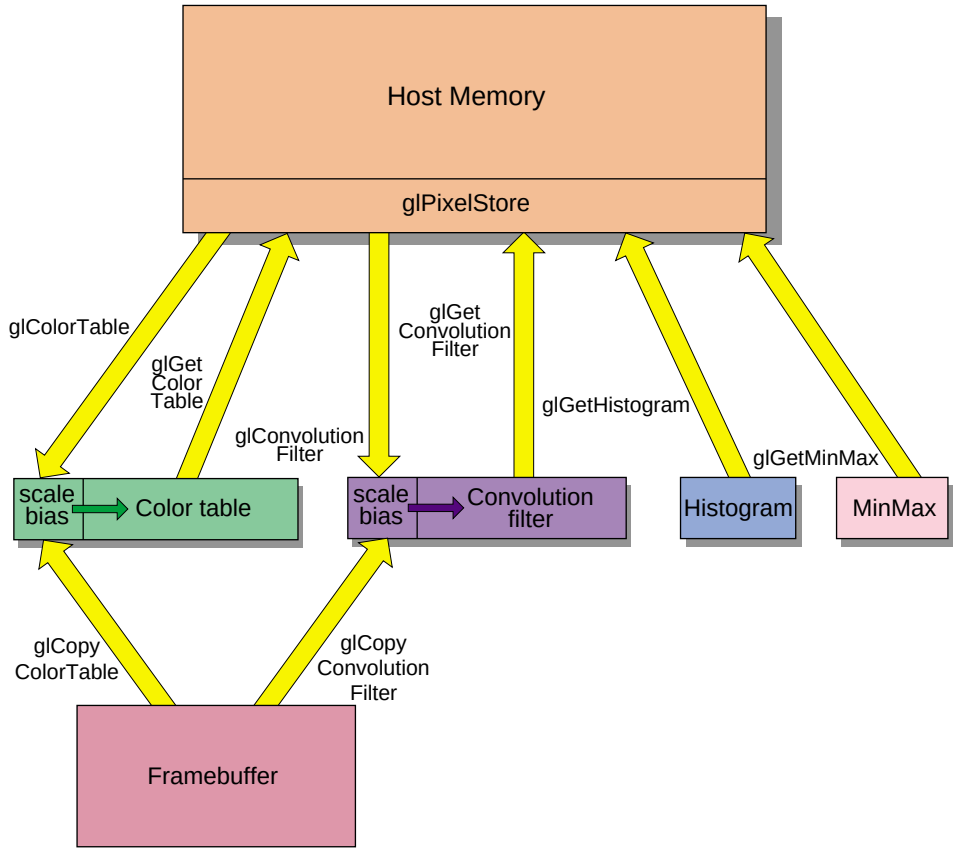


Figure 10-2 Extensions that Modify Pixels During Transfer

Convolution, Histogram, and Color Table in the Pipeline

Figure 10-3 shows 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 10-3 below) 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.*

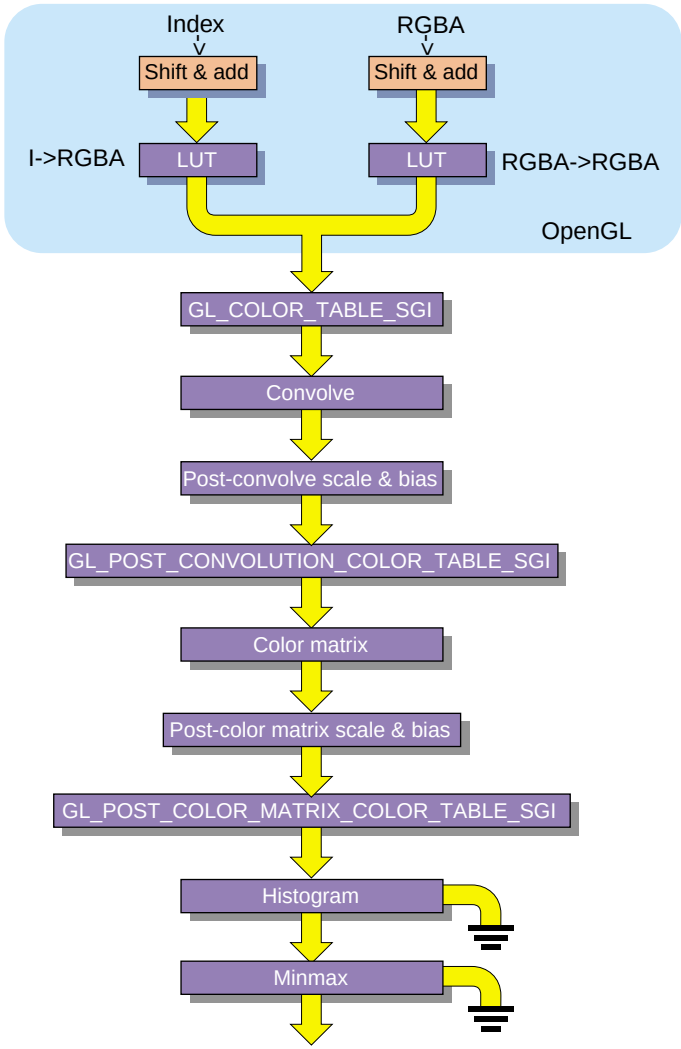


Figure 10-3 Convolution, Histogram, and Color Table in the Pipeline

Interlacing and Pixel Texture in the Pipeline

Figure 10-4 shows where interlacing (see “SGIX_interlace—The Interlace Extension” on page 280) and pixel texture (see “SGIX_pixel_texture—The Pixel Texture Extension” on

page 282) 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.

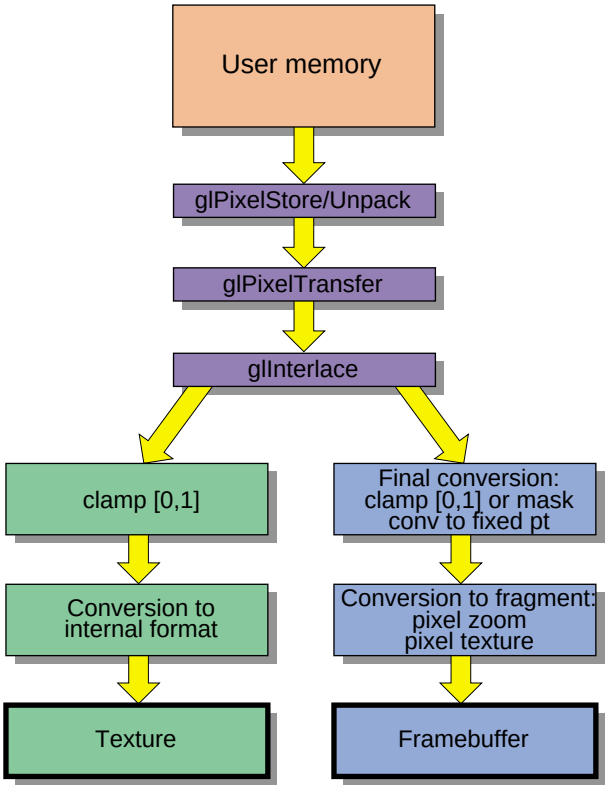


Figure 10-4 Interlacing and Pixel Texture in the Pixel Pipeline

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 10-5.

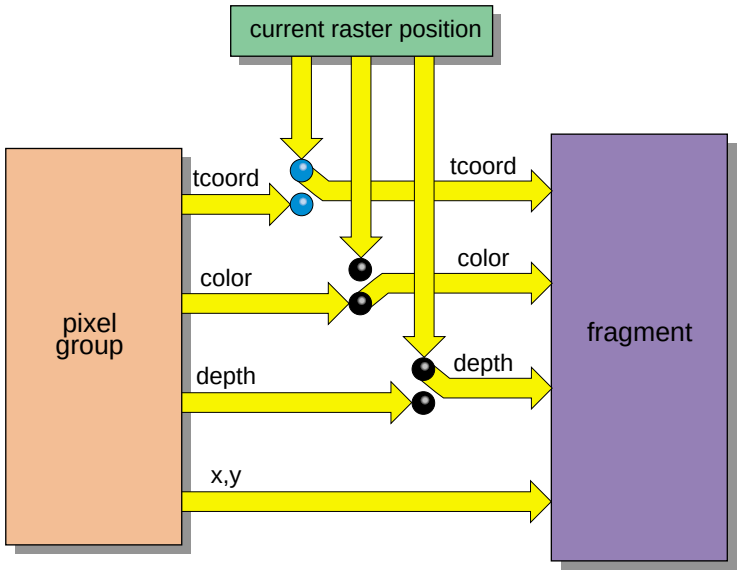


Figure 10-5 Conversion to Fragments

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” on page 282) introduces the switch, highlighted in blue (lighter-colored balls), which provides a way to retrieve the fragment’s texture coordinates from the pixel group. The pixel texture extension also allows you 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 associated with the pixel transfer modes (see Chapter 8, “Drawing Pixels, Bitmaps, Fonts, and Images,” of the *OpenGL Programming Guide).* In general, the following operations are affected:

- All functions that draw and copy pixels or define texture images
- All functions 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. This is the most convenient way to use an ABGR source image with OpenGL.

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

The following code fragment illustrates the use of the extension:

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

{
    unsigned char *image;

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

EXT_convolution—The Convolution Extension

The convolution extension, EXT_convolution, allows you to filter images (for example, to sharpen or blur the) 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 10-6 shows the equations for general convolution at the top and for separable convolution at the bottom.

$$Dest[i,j] = \sum_{l=0}^{Kh} \sum_{k=0}^{Kw} Kernel[k,l]\, Source[i+k, j+l]$$

$$Dest[i,j] = \sum_{l=0}^{Kh} Vert[l] \sum_{k=0}^{Kw} Horiz[k]\, Source[i+k, l+j]$$

Figure 10-6 Convolution Equations

Performing Convolution

Performing convolution consists of the following 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, as shown in the following 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 the man page for **glConvolutionParameterEXT()**.

`GL_CONVOLUTION_FORMAT_EXT`
: Current internal format. For lists of allowable formats, see the man pages for **glConvolutionFilter*EXT()** and **glSeparableFilter2DEXT()**.

`GL_CONVOLUTION_FILTER_{BIAS, SCALE}_EXT`
: Current filter bias and filter scale factors. The value *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 format:

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

The parameters are defined as follows:

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
: Points to two one-dimensional images in memory, is defined by *format* and *type*, is *width* pixels wide.

column
: Points to two one-dimensional images in memory, is defined by *format* and *type*, and 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

The EXT_convolution extension introduces the following 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 10-7 illustrates how the histogram extension collects information for one of the color components. The histogram has the number of bins specified at creation, and information is then collected about the number of 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.

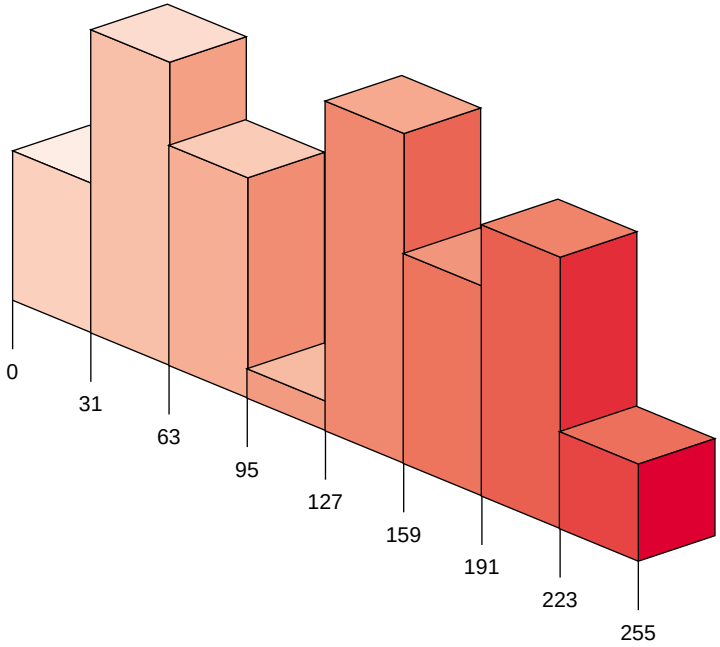


Figure 10-7 How the Histogram Extension Collects Information

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, as shown in the following example:

```
glHistogramEXT(GL_HISTOGRAM_EXT,
               256              /* width (number of bins) */,
               GL_LUMINANCE     /* internalformat */,
               GL_TRUE          /* sink */);
```

   The parameters are defined as follows:

   width ,Specifies the number of histogram entries. Must be a power of 2.

   internalformat Specifies the format of each table entry.

   sink Specifies whether pixel groups are consumed by the histogram operation (`GL_TRUE`) or passed further down the image pipeline (`GL_FALSE`).

2. Enable histogramming by calling

```
glEnable(GL_HISTOGRAM_EXT)
```

3. Perform the pixel operations for which you want to collect information (drawing, reading, 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.

4. Call **glGetHistogramEXT()**, whose format follows, to query the current contents of the histogram:

```
void glGetHistogramEXT( GLenum target, GLboolean reset, GLenum format,
                        GLenum type, GLvoid *values )
```

   The parameters are defined as follows:

   target Must be `GL_HISTOGRAM_EXT`.

   reset Must be `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 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()**, whose format follows:

   `void glMinmaxEXT( GLenum` *target*`, GLenum` *internalformat*`, GLboolean` *sink*`)`

   The parameters are defined as follows:

   target Specifies the table in which the information about the image is to be stored. The value for *target* must be `GL_MINMAX_EXT`.

   internalformat Specifies the format of the table entries. It must be an allowed internal format. See the man page for **glMinmaxEXT()**.

sink Determines whether processing continues. GL_TRUE or GL_FALSE are the valid values. If set to 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 the following function:

```
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()**, whose format follows:

```
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

The EXT_histogram extension introduces the following 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.

Using Packed Pixels

To use packed pixels, provide one of the types listed in Table 10-1 as the *type* parameter to **glDrawPixels()**, **glReadPixels()**, and so on.

Table 10-1 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; the extension multiplies the 4 x 4 color matrix on top of the stack with the components of each pixel. The stack is manipulated using the OpenGL 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.

The following 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; 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 does not replace the color lookup tables provided by the color maps described in the *OpenGL Programming Guide* but provides the following additional lookup capabilities:

- Unlike pixel maps, the color table extension's download operations go through the **glPixelStore()** unpack operations in the same 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 10-3), 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()**, whose format follows:

```
void glColorTableSGI( GLenum target, GLenum internalformat, GLsizei width,
                      GLenum format, GLenum type,const GLvoid *table)
```

The parameters are defined as follows:

target Must be `GL_COLOR_TABLE_SGI`, `GL_POST_CONVOLUTION_COLOR_TABLE_SGI`, or `GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI`.

internalformat Specifies 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	Specifies 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 1D image to which *table* points 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()** with the following parameters:

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 1D 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 and leaves 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 the following lookup tables exist at different points in the image pipeline (see Figure 10-3):

`GL_COLOR_TABLE_SGI`
: Located immediately after index lookup or RGBA to RGBA mapping, and immediately before the convolution operation.

`GL_POST_CONVOLUTION_COLOR_TABLE_SGI`
: 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`
: 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 its associated function.

New Functions

The SGI_color_table extension introduces the following 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 10-4 illustrates 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:

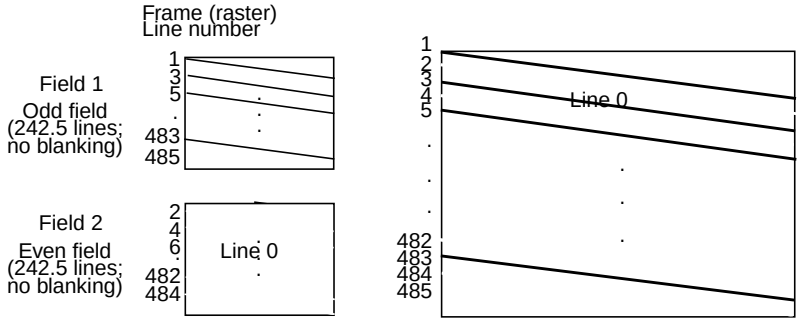


Figure 10-8 Interlaced Video (NTSC, Component 525)

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

Applications that use the extension usually first copy the first set of rows and 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 *cap* set to `GL_INTERLACE_SGIX`.
2. Set the current raster position to xr, yr, as follows:

```
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I0);
```

3. Copy pixels into texture memory (usually field 1 is first), as follows:

```
glCopyTexSubImage2D (GL_TEXTURE_2D, level, xoffset, yoffset, x, y,
                     width, height)
```

4. Set raster position to (xr,yr+zoomy), as follows:

```
glDrawPixels(width, height, GL_RGBA, GL_UNSIGNED_BYTE, I1);
```

5. Copy the pixels from the second field (usually F1 is next). For this call, set the following:

```
y offset += yzoom
y += height (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 10-4 illustrates 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 do not 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, as follows:

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

   This texture does not have to be a 3D texture.

2. Enable pixel texture and begin processing images, as follows:

   ```
   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 10-5 illustrates 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()** function. 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*()** functions. To do this, use this function:

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

The valid values of *mode*, shown in the following, 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()** function. 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 will be derived 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 the following section "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

Pixel texture is supported only on Fuel and InfinitePerformance systems. For further restrictions on the implementation, see your platform release notes and the man page for **glPixelTexGenSGIX()**. For new applications targeting Onyx4 and Silicon Graphics Prism systems, you can achieve similar functionality by writing fragment programs using the fragment color components as texture coordinates.

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

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

The SGIX_pixel_texture extension introduces the function **glPixelTexGenSGIX()**.

Video Extensions

Chapter 6, “Resource Control Extensions,” describes a set of GLX extensions that can be used to control resources. This chapter provides information on the following set of GLX extensions that support video functionality:

SGI_swap_control—The Swap Control Extension

Provided the time required to draw each frame can be bounded, the swap control extension, SGI_swap_control, allows applications to display frames at a regular rate . The extension allows an application to set a minimum period for buffer swaps, counted in display retrace periods.

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

```
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()**.

The functioin **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 affect on that buffer swap.

New Functions

The SGI_swap_control extension introduces the function **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 an 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 man page for `gfxinit` 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()**, whose format follows, to put the current process to sleep until the specified retrace counter:

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

The parameters are defined as follows:

divisor, remainder	The function **glXWaitVideoSyncSGI()** puts the calling process to sleep until the value of the vertical retrace counter (*count*) modulo *divisor* equals *remainder.*
count	This is a pointer to the variable that receives the value of the vertical retrace counter when the calling process wakes up.

New Functions

The SGI_video_sync extension introduces the following functions:

- **glXGetVideoSyncSGI()**
- **glXWaitVideoSyncSGI()**

SGIX_swap_barrier—The Swap Barrier Extension

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

The swap barrier extension, SGIX_swap_barrier, allows applications to synchronize the buffer swaps of different swap groups—that is, on different machines. For information on swap groups, see “SGIX_swap_group—The Swap Group Extension” on page 292.

Why Use the Swap Barrier Extension?

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 cannot 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.

You can use the swap barrier extension through the OpenGL Performer API rather than calling the 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" on page 291). 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 the Swap Ready ports of their graphics pipeline. 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 format:

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

The function **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)
```

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

The function **glXQueryMaxSwapBarriersSGIX()** returns `GL_TRUE` if it succeeds 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

The SGI_swap_barrier extension introduces the following 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. Because the two **glXSwapBuffers()** calls are not atomic, the following cases are possible:

- 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. Joining the *left_window* and *right_window* into a swap group ensures that the windows swap together atomically. This could be done during initialization by making the following call:

```
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 making the following call:

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

Swap Group Details

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

```
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 its swap group, 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 both of 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.

The function **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" on page 289).

New Function

The SGIX_swap_group extension introduces the function **glJoinSwapGroupSGIX()**.

SGIX_video_resize—The Video Resize Extension

Note: This extension is only supported on InfiniteReality systems.

The video resize extension, SGIX_video_resize, is an extension to GLX that allows the framebuffer 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 framebuffer 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 format:

```
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.

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" on page 295).

   The function has the following format:

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

   The parameters are defined as follows:

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 systems support multiple output channels (two or eight depending on the Display Generator board type). Each channel can be dynamically resized independently.

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 format:

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

   The parameters are defined as follows:

   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 Specify the precision deltas for the origin and size of the area specified by **glXChannelRectSGIX()**.

4. Call **XSGIvcQueryChannelInfo()** (an interface to the X video control 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:
   - Based on performance requirements, determine the area that will be drawn. 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 systems, 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 format:

```
int glXChannelRectSGIX( Display *display, int screen,
              int channel, int x, int y, int w, int h)
```

The parameters are defined as follows:

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—Specify 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 man page for **glxChannelRectSGIX()** illustrates how to use the extension:

Example 11-1 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 limited..
```

```
    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

The SGIX_video_resize extension introduces the following functions:

- **glXBindChannelToWindowSGIX()**
- **glXChannelRectSGIX()**
- **glXChannelRectSyncSGIX()**
- **glXQueryChannelRectSGIX()**

Miscellaneous OpenGL Extensions

This chapter explains how to use several extensions that are not easily grouped with texturing, imaging, or GLX extensions. Example code is provided as needed. The following extensions are described:

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 function:

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

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

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 a *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.The *userData* is a copy of the pointer that was specified at the last call to **gluNurbsCallbackDataEXT()**.

The callbacks have the following formats:

```
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 12-1 provides additional information on each callback.

Table 12-1 NURBS Tessellator Callbacks and Their Description

Callback	Description
GLU_NURBS_BEGIN_EXT GLU_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.
GLU_NURBS_VERTEX_EXT GLU_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.
GLU_NURBS_NORMAL_EXT GLU_NURBS_NORMAL_DATA_EXT	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 you provide 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.
GLU_NURBS_COLOR_EXT GLU_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 you provide a color map (GL_MAP1_COLOR_4 or GL_MAP2_COLOR_4). The *color* value contains four components: R, G, B, or A.The default **color()** and **colorData()** callback functions are NULL.

Table 12-1 NURBS Tessellator Callbacks and Their Description **(continued)**

Callback	Description
GLU_NURBS_TEXCOORD_EXT GLU_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.
GLU_NURBS_END_EXT GLU_NURBS_END_DATA_EXT	Is invoked at the end of a primitive. The default **end()** and **endData()** callback functions are NULL.
GLU_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 of the following:

- GLU_PATH_LENGTH
- GLU_PARAMETRIC_ERROR
- GLU_DOMAIN_DISTANCE
- GLU_OBJECT_PATH_LENGTH_EXT
- 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 12-2 contrasts the methods provided by the extension with the existing methods.

Table 12-2 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.

Table 12-2 Tessellation Methods **(continued)**

Method	Description
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

Note: This extension is only supported on Fuel, Tezro, InfinitePerformance, and InfiniteReality systems.

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 format:

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

The parameters are defined as follows:

list	The display list
pname	`GL_LIST_PRIORITY_SGIX`
params	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 **glGetListParameterivSGIX()**, which has the following format:

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

The parameters are defined as follows:

list	The display list
pname	`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

The SGIX_list_priority extension introduces the following functions:

- **glListParameterSGIX()**
- **glGetListParameterSGIX()**

SGIX_instruments—The Instruments Extension

Note: This extension is only supported on InfiniteReality systems.

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

Why Use 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 the following:

  – 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 functions 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 describes 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 format:

```
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" on page 311.)

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 formats:

```
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 the following 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 man page for **glInstrumentsSGIX()**. Currently, only InfiniteReality systems support any instruments.

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 format:

```
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 *markerp*. 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 format:

```
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 12-1 provides pseudo code for using the instruments extension.

Example 12-1 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), buffer);

             /* enable the instruments from which to take measurements */
             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 was 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 PollInstrumentsSGIX.
              */
             count0 = glGetInstrumentsSGIX();
             /* Since 0, 1, or 2 measurements might have been returned 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 first
     * measurement request (should be MARKER1).  While it is 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 size in GLints
     * of the records returned for all instruments that we have enabled.
     */
}
#endif
```

New Functions

The SGIX_instruments extension introduces the following functions:

glInstrumentsBufferSGIX()

glStartInstrumentsSGIX()

glStopInstrumentsSGIX()

glReadInstrumentsSGIX()

glPollInstrumentsSGIX()

glGetInstrumentsSGIX()

Vertex and Fragment Program Extensions

In addition to many extensions to the classical fixed-function OpenGL rendering pipeline, Onyx4 and Silicon Graphics Prism systems support the following extensions for vertex and fragment programs:

- ARB_vertex_program
- ARB_fragment_program

Collectively, *vertex* and *fragment programs* are referred to as graphics pipeline programs or just *pipeline programs*.

These extensions allow applications to replace most of the normal fixed-function transformation, lighting, rasterization, and texturing operations with application-defined programs that execute on the graphics hardware. The extensions enable a nearly unlimited range of effects previously available only through offline rendering or by multipass fixed-function algorithms.

This chapter describes how to define and use vertex and fragment programs and includes an overview of the programming language in which these programs are specified. This chapter also briefly describes the following obsolete (legacy) vertex and fragment program extensions supported only for compatibility:

- ATI_fragment_shader
- EXT_vertex_shader

The structure of this chapter differs from that of the other chapters that describe extensions because of the level of detail given to programming the vertex and fragament programs. This chapter uses the following structure:

The Vertex and Fragment Program Extensions

The ARB_vertex_program and ARB_fragment_program extensions allow applications to replace respectively the fixed-function vertex processing and fragment processing pipeline of OpenGL 1.3 with user-defined programs.

Why Use Pipeline Programs?

The fixed-function rendering pipeline of OpenGL 1.3 together with the wide range of OpenGL extensions supported by Onyx4 and Silicon Graphics Prism systems is very flexible, but the achievable rendering effects are constrained by the hardwired algorithms of the fixed-function pipeline. If an application needs to use a custom lighting model, to combine multiple textures in a way not expressable by register combiners, or to do anything else that is difficult to express within the fixed-function pipeline; it should consider if the desired effect can be expressed as a vertex and/or fragment program.

While pipeline programs are not yet expressable in a fully general-purpose, Turing-complete language, the limited programmability provided is more than adequate for many advanced rendering algorithms. The capabilities of pipeline programs are rapidly growing as more general-purpose languages are supported by graphics hardware.

Alternatives to Pipeline Programs

Before pipeline programs, the most common way to implement advanced rendering algorithms, while still taking advantage of graphics hardware, was to decompose the algorithm into a series of steps, each expressable as a single rendering pass of the fixed-function OpenGL pipeline. By accumulating intermediate results in pixel buffers, aux buffers, or textures, very complex effects could be built up by such *multipass rendering*.

This approach is widely used in older programs and in languages such as the SGI OpenGL Shader, a compiler which turns a high-level shading language program into a equivalent series of fixed-function rendering passes.

The disadvantages of multipass rendering are the following:

- Performance

  Multiple rendering passes usually require re-transforming geometry for each pass. The multiple passes consume additional CPU-to-graphics bandwidth for re-copying geometry and consume additional graphics memory and bandwidth for storing intermediate results; all of these requirements reduce performance.

- Complexity

  Converting a complex rendering algorithm into multiple fixed-function passes can be a tedious task that requires a deep understanding of the capabilities of the graphics pipeline. The meaning of the resulting passes is difficult to infer even with knowledge of the algorithm. Also, restructuring applications to perform multipass rendering is often necessary. While software like the SGI OpenGL Shader can assist in these steps, it is still less obvious to do complex multipass rendering than to simply express the algorithm as a single vertex or fragment program.

- Accuracy

  The accuracy achievable with multipass rendering is constrained by the limited precision of the intermediate storage (for example, pixel buffers, aux buffers, textures, etc.) used to accumulate intermediate results between passes. Typically, the internal precision of the vertex and fragment processing pipelines is much higher than the external precision (8–12 bits/color component) in which intermediate data can be stored. Errors are generated when clamping intermediate data to external precision, and those errors can rapidly accumulate in the later rendering passes.

For all these reasons, pipeline programs are the preferred way of expressing rendering algorithms too complex to fit in a single fixed-function rendering pass. However, in many cases, the fixed-function pipeline is still more than adequate for application needs. You must also be cautious because current graphics hardware only supports pipeline programs of a limited length and complexity and because performance may degrade rapidly if certain types of programmable operations are combined or expressed in the wrong order.

Using Pipeline Programs

The vertex and fragment program extensions are much more complicated than most fixed-function OpenGL extensions. This section describes the extensions in the following subsections:

Managing Pipeline Programs

Pipeline programs are represented by object names (of type `GLuint`) that are managed in exactly the same fashion as texture and display list names with the following routines for allocating unused program names, deleting programs, and testing if a name refers to a valid program:

```
void glGenProgramsARB(GLsizei n, GLuint *programs);
void glDeleteProgramsARB(GLsizei n, const GLuint *programs);
GLboolean glIsProgramARB(GLuint program);
```

Binding Programs

To bind a program name as the currently active vertex or fragment program, make the following call:

```
void glBindProgramARB(GLenum target, GLuint programs);
```

Set the argument *target* to `GL_VERTEX_PROGRAM_ARB` or `GL_FRAGMENT_PROGRAM_ARB`. Similar to texture objects, there is a **default program name** of 0 bound for each type of program in the event that the application does not bind a generated name.

Defining and Enabling Programs

To define the contents of a vertex or fragment program for the currently bound program name, make the following call:

```
void glProgramStringARB(GLenum target, GLenum format, GLsizei length, const
GLvoid *string);
```

The arguement values are defined as follows:

target	Has the same meaning as for **glBindProgramARB()**.
format	Specifies the encoding of the program string and must be `GL_PROGRAM_FORMAT_ASCII_ARB`, indicating a 7-bit ASCII character string.
string	Contains the program string. If *string* is a valid program (as described in section "Structure of Pipeline Programs" on page 319), the program bound to *target* will be updated to execute the program when the corresponding target is enabled.
length	Specifies the length of the string. Because the length is specified in the call, *string* need not have a trailing `NULL` byte, unlike most C language strings.

To use the currently bound vertex or fragment program (substituting it for the corresponding fixed functionality, as described in the next section) or to return to using the fixed-function pipeline, call **glEnable()** or **glDisable()**, respectively, with parameters `GL_VERTEX_PROGRAM_ARB` or `GL_FRAGMENT_PROGRAM_ARB`.

How Programs Replace Fixed Functionality

Vertex programs substitute for the following OpenGL fixed vertex processing functionality:

- Modelview and projection matrix vertex transformations
- Vertex weighting and blending (if the ARB_vertex_blend extension is supported)
- Normal transformation, rescaling, and normalization
- Color material
- Per-vertex lighting
- Texture coordinate generation and texture matrix transformations
- Per-vertex point size computations (if the ARB_point_parameters extension is supported)
- Per-vertex fog coordinate computations (if the EXT_fog_coord extension is supported)
- User-defined clip planes
- Normalization of `GL_AUTO_NORMAL` evaluated normals
- All of the preceding functionality when computing the current raster position

The following fixed vertex processing functionality is always performed even when using vertex programs:

- Clipping to the view frustum
- Perspective divide (division by w)
- The viewport transformation
- The depth range transformation
- Front and back color selection (for two-sided lighting and coloring)
- Clamping the primary and secondary colors to [0,1]
- Primitive assembly and subsequent operations
- Evaluators (except for `GL_AUTO_NORMAL`)

Fragment programs substitute for the following OpenGL fixed fragment processing functionality:

- Texture application (including multitexture, texture combiner, shadow mapping, and any other fixed-function texturing extensions)
- Color sum (if the EXT_secondary_color extension is supported)
- Fog application

Structure of Pipeline Programs

Both vertex and fragment programs are expressed in a low-level, register-based language similar to a traditional CPU assembler language. However, the registers are four-element vectors, supporting vector data types such as homogeneous coordinates (X, Y, Z, W components) and colors (R, G, B, A components). The instruction sets for both types of programs are augmented to support common mathematical and graphical operations on these four-element vectors.

A pipeline program has the following structure:

program-type
statement1
statement2
.
.
.
statementn
`END`

For vertex programs, *program-type* must be `!!ARBvp1.0`. For fragment programs, it must be `!!ARBfp1.0`.

Statements may be one of the following:

- Program options
- Naming statements
- Program instructions

The statements must be terminated by semicolons (;). Whitespace (spaces, tabs, newlines, and carriage returns) is ignored, although programs are typically written with one

statement per line for clarity. Comments, which are ignored, are introduced with the # character and continue to the next newline or carriage return.

When executing programs, instructions are processed in the order they appear in the program string. There are no looping or branching constructs in either vertex or fragment programs.

Program Options

Statements that control extended language features are called option statements. The following is an example:

```
# Vertex program is position-invariant
OPTION ARB_position_invariant;
```

The following are the currently defined program options:

- Fog application options (fragment programs only)
- Precision hint options (fragment programs only)
- Position-Invariant option (vertex programs only)

Future OpenGL extensions may introduce additional program options; such options are only valid if the corresponding extension is supported by the implementation.

Fog Application Options (fragment programs only)

These options allow use of the OpenGL fixed-function fog model in a fragment program without explicitly performing the fog computation.

If a fragment program specifies one of the options `ARB_fog_exp`, `ARB_fog_exp2`, or `ARB_fog_linear`, the program will apply fog to the program's final clamped color output using a fog mode of `GL_EXP`, `GL_EXP2`, or `GL_LINEAR`, respectively.

Using fog in this fashion consumes extra program resources. The program will fail to load under the following conditions:

- You specify a fog option and the number of temporaries the program contains exceeds the implementation-dependent limit minus one.
- You specify a fog option and the number of attributes the program contains exceeds the implementation-dependent limit minus one.

- You specify a fog option and the number of parameters the program contains exceeds the implementation-dependent limit minus two.
- You specify the `ARB_fog_exp` option and the number of instructions or ALU instructions the program contains exceeds the implementation-dependent limit minus three.
- You specify the `ARB_fog_exp2` option and the number of instructions or ALU instructions the program contains exceeds the implementation-dependent limit minus four.
- You specify the `ARB_fog_linear` option and the number of instructions or ALU instructions the program contains exceeds the implementation-dependent limit minus two.
- You specify more than one of the fog options.

Precision Hint Options (fragment programs only)

Fragment program computations are carried out at an implementation- dependent precision. However, some implementations may be able to perform fragment program computations at more than one precision and may be able to trade off computation precision for performance.

If a fragment program specifies the `ARB_precision_hint_fastest` program option, implementations should select precision to minimize program execution time with possibly reduced precision. If a fragment program specifies the `ARB_precision_hint_nicest` program option, implementations should maximize the precision with a longer execution time.

Only one precision control option may be specified by any given fragment program. A fragment program that specifies both the `ARB_precision_hint_fastest` and `ARB_precision_hint_nicest` program options will fail to load.

Position-Invariant Option (vertex programs only)

If a vertex program specifies the `ARB_position_invariant` option, the program is used to generate all transformed vertex attributes, except for position. Instead, clip coordinates are computed, and user clipping is performed as in the fixed-function OpenGL pipeline. Use of position-invariant vertex programs should be used when the transformed position of a vertex will be the same whether vertex program mode is enabled or fixed-function vertex processing is performed. This allows mixing both types of vertex processing in multipass rendering algorithms.

When the position-invariant option is specified in a vertex program, vertex programs are not allowed to produce a transformed position. Therefore, `result.position` may not be bound or written by such a program. Additionally, the vertex program will fail to load if the number of instructions it contains exceeds the implementation-dependent limit minus four.

Naming Statements

Statements that associate identifiers with attributes, parameters, temporaries, or program output are called naming statements. The following are the six types of naming statements:

- Attribute statements
- Parameter statements
- Temporary statements
- Address statements
- Alias statements
- Output statements
- Attribute Statements

Attribute statements bind an identifier to a vertex or fragment attribute supplied to the program. Attributes are associated with the particular vertex or fragment being processed, and their values typically vary for every invocation of a program. They are defined in OpenGL through commands such as **glVertex3f()** or **glColor4ub()**, or in the case of fragments, generated by vertex processing.

A few examples of attributes are vertex position, color, and texture coordinates; or fragment position, color, and fog coordinate. Section “Vertex and Fragment Attributes” on page 329 provides a complete list of vertex and fragment attributes. The following are examples of attribute statements:

```
# Bind vertex position (e.g. glVertex) to attribute ‘position’
ATTRIB position = vertex.position;
#
# Bind fragment texture coordinate set one to attribute ‘texcoord’
ATTRIB texcoord = fragment.texcoord[1];
```

Attributes are read-only within a program.

Parameter Statements

Parameter statements bind an identifier to a program parameter. Parameters have the following four types:

- Program environment parameters

  Constants that are shared by all vertex programs.

- Program local parameters

  Constants that are restricted to a single vertex or fragment program.

- OpenGL state values

  Items such as transformation matrices, or lighting, material, and texture coordinate generation parameters.

- Constants declared within a program

  Comma-delimited lists of one to four values enclosed in braces. If fewer than four values are specified, the second, third, and fourth values default to `0.0`, `0.0`, and `1.0`, respectively; or they are single values not enclosed in braces, in which case all four components of the parameter are initialized to the specified value.

Parameter statements may also be declared as arrays, which are initialized from subranges of program parameters or state, which are themselves arrays.

Section "Vertex and Fragment Program Parameters" on page 333 provides a complete list of program environment parameters, program local parameters, and OpenGL state that may be used as parameters. The following are examples of parameter statements:

```
# 'var' is bound to program environment parameter 1
     PARAM var = program.env[1];

     # 'vars' is bound to program environment parameters 0-3
     PARAM vars[4] = program.env[0..3];

     # 'lvar' is bound to program local parameter 2
     PARAM lvar = program.local[2];

     # 'ambient' is bound to the ambient color of light 1
     PARAM ambient = state.light[1].ambient;

     # 'cplane' is bound to the coefficients of user clip plane 0
     PARAM cplane = state.clip[0].plane;
```

```
# ‘coeffs’ is bound to the four constant values -1,0 1.0, e, pi
PARAM coeffs = { -1.0, 1.0, 2.71828, 3.14159 };

# ‘ones’ is bound to the constant values 1.0, 1.0, 1.0, 1.0
PARAM ones = 1.0;
```

Parameters are read-only within a program.

Temporary Statements

Temporary statements declare temporary variables; these are read/write registers used only within a single execution of a program. Initially, the contents of temporaries are undefined. Temporaries are declared as in the following examples:

```
# Declare a single temporary
TEMP temp1;

# Declare multiple temporaries in a single statement
TEMP temp2, temp3;
```

The maximum number of temporaries that can be declared in a single program is implementation-dependent and is described further in section “Program Resource Limits and Usage” on page 372.

Address Statements

Address statements declare address registers; these are read/write registers used only within a single execution of a vertex program and allow a form of indirect accessing into parameter arrays. Address statements are only supported in vertex programs.

Address registers are declared either singly or a in multiple fashion like temporaries but using the `ADDRESS` statement, as in the following example:

```
# Declare two address registers ‘addr’ and ‘index’
ADDRESS addr, index;
```

Only the first component of an address register (`.x`) is used. For an address register `addr`, this component is referred to as `addr.x`. Section “Program Instructions” on page 326 further describes register component selection. As shown in the following example, address registers are loaded with the `ARL` command:

```
# Load address register with the 2nd component (.y) of temporary temp0
ARL addr.x, temp0.y;
```

The value loaded is converted to an integer by clamping towards negative infinity.

Given a parameter array and an address register, a particular element of the array can be selected based on the address register by using the subscript notation [`addr.x`+*offset*], where *offset* is a value in the range –64..63. The following example illustrates the use of the subscript notation:

```
# Params is bound to the first 8 elements of the program local
# parameters.
PARAM params[8] = program.local[0..7];

# Move parameter at index addr.x+2 into register temp0
MOV temp0, params[addr.x+2];
```

Alias Statements

Alias statements declare identifiers that are defined to have the same meaning as another already declared identifier of any type. They do not count towards program resource limits. For example, a temporary can be aliased as follows:

```
# Declare temporary ‘temp0’
TEMP temp0;

# Declare alias ‘alias’ for temp0. ‘alias’ and ‘temp0’ may be used
# interchangably
ALIAS alias = temp0;
```

Output Statements

Output statements declare identifiers that bind to program output. Output depends on the type of program.

For vertex programs, output includes values such as transformed vertex position, primary and secondary colors, transformed texture coordinates, which are passed on to rasterization. After rasterization, interpolated results of this output are available as attributes of fragment programs or are used in fixed-function fragment processing in place of the attributes resulting from fixed-function vertex processing.

For fragment programs, output includes color(s) and depth values, which are passed on to raster processing in place of the colors and depths generated by fixed-function fragment processing. Section “Pipeline Program Input and Output” on page 329 describes program output further.

The following are examples of output statements:

```
# Bind vertex output position (e.g., transformed vertex coordinates)
# to register 'windowpos'
OUTPUT windowpos = result.position;

# Bind fragment output depth (e.g., Z value) to register 'depth'
OUTPUT depth = result.depth;
```

Output is write-only within a program.

Program Instructions

Pipeline program instructions are either four-element vector or scalar operations performed on one, two, or three source operands and one destination operand. The operands may be either attribute, parameter, or temporary registers. The general format of instructions is one of the following:

mnemonic dstreg,srcreg1

mnemonic dstreg,srcreg1,srcreg2

mnemonic dstreg,*srcreg1,srcreg2*,*srcreg3*

The fields are defined as follows:

mnemonic	The instruction name
dstreg	The destination register name
srcregi	Source register names

Section "Program Instruction Summary" on page 351 provides a complete list of instructions supported by vertex and fragment programs.

Scalar Component Selection

When a scalar source operand is required, identify it by appending one of `.x`, `.y`, `.z`, or `.w` to the register name to select the first, second, third, or fourth components, respectively, of the source register. These selectors are intended to refer to the X, Y, Z, and W components of a register being used as a XYZW vector. The following example computes the cosine of the second component of the source register `coord`:

```
COS result, coord.y;
```

In fragment programs, but not in vertex programs, the selectors `.r`, `.g`, `.b`, and `.a` may be used interchangably with the corresponding `.x`, `.y`, `.z`, and `.w` selectors. These selectors are intended to refer to the red, green, blue, and alpha components of a register being used as an RGBA color. The following example computes the base 2 logarithm of the fourth component of the source register `color`:

```
LG2 result, color.a;
```

Vector Component Negation and Swizzling

Any source register may be modified by prepending a minus sign (-) to the register name. Each component is negated and the resulting vector used as input to the instruction. For example, the following two statements are equivalent:

```
# Compute result = src0 - src1
SUB result, src0, src1;
# Compute result = src0 + (-src1) = src0 - src1
ADD result, src0, -src1;
```

In addition, components of a source register may be arbitrarily selected and reordered before being used as input to an instruction. This operation is called *swizzling*. To swizzle a source register, append a four-letter suffix of `.????` to the register name, where each `?` may be one of the component selectors `x`, `y`, `z`, or `w`. In fragment programs, but not in vertex programs , the selectors `r`, `g`, `b`, or `a` may also be used.

The selectors map components of the source register; the first, second, third, and fourth selectors determine the source of the first, second, third, and fourth components, respectively, of the actual register value passed to the instruction. For example, the following code reverses the components of a register:

```
PARAM src = { 1.0, 2.0, 3.0, 4.0 };
TEMP result;
MOV result, src.wzyx;
# result now contains { 4.0, 3.0, 2.0, 1.0 }
```

Swizzling may copy a component of the source register into multiple components of the instruction input by replicating selectors. For example, the following code replicates the first and third components of a register:

```
PARAM src = { 1.0, 2.0, 3.0, 4.0 };
TEMP result;
MOV result, src.xxzz;
# result now contains { 1.0, 1.0, 3.0, 3.0 }
```

To replicate a single component of a register into all four components of the instruction input, a shorthand notation using a single component selector may be used. The following code is equivalent to replicating the same component selector four times:

```
PARAM src = { 1.0, 2.0, 3.0, 4.0 };
TEMP result;
# src.y is equivalent to src.yyyy
MOV result, src.y;
# result now contains { 2.0, 2.0, 2.0, 2.0 }
```

Destination Write Masking

Program instructions write a four-component result vector to a single destination register. Writes to individual components of the destination register may be controlled by specifying a component *write mask*. To mask a destination register, append a period (.) followed by selectors for the components to be written (between one and four). The selectors must be unique and must appear in the order `xyzw`. In fragment programs, but not in vertex programs, the `rgba` selectors may also be used. For example, the following line writes only the first and third components of a vector and leaves the second and fourth components unchanged:

```
MOV result.xz, src
```

Fragment Program Destination Clamping

In fragment programs, but not in vertex programs, instructions may be modified to *clamp* values to the range `[0,1]` before writing to the unmasked components of a destination register. Clamping is particularly useful when operating in the `[0,1]` color space limits of the output framebuffer, when using texture coordinates, when computing address register offsets, or for other purposes. Fragment program instructions support clamping by appending the suffix `_SAT` to the instruction mnemonic. Clamping the RGB components to `[0,1]` and using write masks to leave the A component of the destination unchanged, the following example copies a color vector:

```
PARAM color = { -0.1, 0.7, 1.2, 1.0 };
TEMP result;
MOV_SAT result.rgb, color;
# result now contains { 0.0, 0.7, 1.0, ??? }
```

Constants

Numeric constants may be used in place of source registers. For instructions requiring scalar input, replace the register name with a single, floating point number. For instructions requiring vector input, replace the register name with a constant vector

defined in the same fashion as constants in parameter statements. The following are examples of scalar and vector constants in instructions:

```
# Compute cosine of constant value 2.0
COS result, 2.0;
# Subtract 1.0 from each element of src
SUB result, src, { 1.0, 1.0, 1.0, 1.0 };
```

Pipeline Program Input and Output

The preceding description of program structure includes mechanisms for binding input to and output from programs. This section describes the complete set of input and output available to pipeline programs. It is important to remember that vertex and fragment programs have different input and output, because they replace different portions of the OpenGL fixed-function pipeline.

The input available to programs includes attributes specific to a vertex or fragment (such as position, color, or texture coordinates) and parameters, which are constant values associated with a single program or collectively with all programs.

The output that is generated by programs are results passed on to later stages of the graphics pipeline, such as transformed vertices and texture coordinates, lit colors, or fragment depth values.

Vertex and Fragment Attributes

This section lists all possible attributes for vertex and fragment programs and includes a description of the component usage and examples of commands creating the corresponding OpenGL state.

Vertex Attributes

Vertex attributes are specified by the application using OpenGL immediate-mode commands such as **glVertex3f()** or vertex array commands such as **glNormalPointer()**. Attributes of a vertex are made available to a vertex program when it is executing for that vertex and can be accessed in instructions either by binding their names with the `ATTRIB` naming statement or directly by use of the attribute name.

In addition to the builtin OpenGL attributes such as position, normal, color, and texture coordinates, vertex programs may be passed additional per-vertex values by using *generic vertex attributes*. Generic attributes are four-component vectors specified using a new set of OpenGL commands. The maximum number of generic attributes supported is implementation-dependent, but must be at least 16. Generic attributes are specified in OpenGL using the commands described in section "Generic Vertex Attribute Specification" on page 348.

In a vertex program, many generic attributes are *aliased* onto builtin OpenGL attributes. When declaring attributes in the program, only one of the builtin attribute or the corresponding generic attribute aliased onto the builtin may be bound. Attempting to bind both a builtin and the corresponding generic attribute results in an error when loading the program. Not all generic attributes have builtin attribute aliases, and conversely so.

Table 13-1 lists the vertex program attributes. In the table, the notation *n* refers to additional implementation-specific resources beyond those explicitly numbered. The possible values of *n* depend on the maximum number of texture coordinate sets, generic vertex attributes, vertex weights, or vertex indices supported by the implementation. For example, if the implementation supports 24 generic vertex attributes, values of *n* for `vertex.attrib`[*n*] range from 0 to 23.

Table 13-1 Builtin and Generic Vertex Program Attributes

Generic Binding	Builtin Binding	Builtin Component Usage	Builtin Description	OpenGL Command
`vertex.attrib[0]`	`vertex.position`	(x,y,z,w)	Object-space vertex position	**glVertex3f()**
`vertex.attrib[1]`	`vertex.weight`	(w,w,w,w)	Vertex weights 0–3	**glWeightfARB()**
`vertex.attrib[1]`	`vertex.weight[`*n*`]`	(w,w,w,w)	Additional vertex weights *n*-*n*+3	**glWeightfARB()**
`vertex.attrib[2]`	`vertex.normal`	(x,y,z,1)	Normal vector	**glNormal3f()**
`vertex.attrib[3]`	`vertex.color`	(r,g,b,a)	Primary color	**glColor4ub()**

Table 13-1 Builtin and Generic Vertex Program Attributes **(continued)**

Generic Binding	Builtin Binding	Builtin Component Usage	Builtin Description	OpenGL Command
`vertex.attrib[3]`	`vertex.color.primary`	(r,g,b,a)	Primary color	**glColor4ub()**
`vertex.attrib[4]`	`vertex.color.secondary`	(r,g,b,a)	Secondary color	**glSecondaryColor3ubEXT()**
`vertex.attrib[5]`	`vertex.fogcoord`	(f,0,0,1)	Fog coordinate	**glFogCoordEXT()**
`vertex.attrib[6]`			Generic attribute 6 (not aliased)	
`vertex.attrib[7]`			Generic attribute 7 (not aliased)	
`vertex.attrib[8]`	`vertex.texcoord`	(s,t,r,q)	Texture coordinate set 0	**glTexCoord3f()**
`vertex.attrib[8]`	`vertex.texcoord[0]`	(s,t,r,q)	Texture coordinate set 0	**glTexCoord3f()**
`vertex.attrib[9]`	`vertex.texcoord[1]`	(s,t,r,q)	Texture coordinate set 1	**glMultiTexCoord(TEXTURE1,...)**
`vertex.attrib[10]`	`vertex.texcoord[2]`	(s,t,r,q)	Texture coordinate set 2	**glMultiTexCoord(TEXTURE2,...)**
`vertex.attrib[11]`	`vertex.texcoord[3]`	(s,t,r,q)	Texture coordinate set 3	**glMultiTexCoord(TEXTURE3,...)**
`vertex.attrib[12]`	`vertex.texcoord[4]`	(s,t,r,q)	Texture coordinate set 4	**glMultiTexCoord(TEXTURE4,...)**
`vertex.attrib[13]`	`vertex.texcoord[5]`	(s,t,r,q)	Texture coordinate set 5	**glMultiTexCoord(TEXTURE5,...)**
`vertex.attrib[14]`	`vertex.texcoord[6]`	(s,t,r,q)	Texture coordinate set 6	**glMultiTexCoord(TEXTURE6,...)**
`vertex.attrib[15]`	`vertex.texcoord[7]`	(s,t,r,q)	Texture coordinate set 7	**glMultiTexCoord(TEXTURE7,...)**

Table 13-1 Builtin and Generic Vertex Program Attributes **(continued)**

Generic Binding	Builtin Binding	Builtin Component Usage	Builtin Description	OpenGL Command
`vertex.attrib[8+`*n*`]`	`vertex.texcoord[`*n*`]`	(s,t,r,q)	Additional texture coordinate sets	**glMultiTexCoord(TEXTURE0+*n*,...)**
	`vertex.matrixindex`	(i,i,i,i)	Vertex matrix indices 0–3	**glMatrixIndexubvARB()**
	`vertex.matrixindex[`*n*`]`	(i,i,i,i)	Additional vertex matrix *n*–*n*+3	**glMatrixIndexubvARB()**
`vertex.attrib[`*n*`]`	`Depends on` *n*	(x,y,z,w)	Additional generic attributes	

Fragment Attributes

Fragment attributes are initially generated by either vertex program output or by the fixed-function OpenGL vertex pipeline if vertex programs are disabled.

Depending on the type of primitive being drawn and on the shading model (`GL_FLAT` or `GL_SMOOTH`) selected, the resulting values may be interpolated on a per-fragment basis during rasterization and fragment generation. Unlike vertex attributes, there are no generic fragment attributes.

Attributes of a fragment are made available to a fragment program when it is executing for that fragment and can be accessed in instructions either by binding their names with the `ATTRIB` naming statement or directly by use of the attribute name.

In Table 13-2, the notation *n* refers to additional implementation-specific texture coordinates beyond those explicitly numbered. The possible values of *n* range from zero up to the maximum number of texture coordinate sets supported by the implementation minus one.

Table 13-2 Fragment Program Attributes

Attribute Binding	Component Usage	Description
`fragment.color`	(r,g,b,a)	Primary color
`fragment.color.primary`	(r,g,b,a)	Primary color
`fragment.color.secondary`	(r,g,b,a)	Secondary color
`fragment.texcoord`	(s,t,r,q)	Texture coordinate set 0
`fragment.texcoord[`*n*`]`	(s,t,r,q)	Texture coordinate set *n*
`fragment.fogcoord`	(f,0,0,1)	Fog distance/coordinate
`fragment.position`	(x,y,z,1/w)	Window position

Vertex and Fragment Program Parameters

Parameters are additional values made available during the execution of programs. When rendering a single primitive, such as a triangle, the vertex and fragment attribute values will differ for every vertex making up the triangle and for every fragment generated by triangle rasterization. However, parameter values will be the same for every vertex and for every fragment.

As cited earlier, the following are the four types of parameters:

- Program environment parameters

  Shared by all programs of a particular type; that is, there is one set of environment parameters for vertex programs and a different set for fragment programs.

- Program local parameters

  Specific to a single bound program.

- OpenGL state values

  Items such as matrices, and material and light properties are available.

- Constants

  Special cases of program local parameters

Program environment and local parameters are four-component vectors specified using a new set of OpenGL commands described in section "Program Parameter Specification" on page 347. The maximum number of parameters supported is implementation-dependent but must be at least 96 each for the vertex program environment and program locals and 24 each for the fragment program environment and program locals. Constants may be specified otherwise.

Program Environment and Local Parameters

Program parameters can be accessed in instructions either by binding their names with the PARAM naming statement or directly by use of the parameter name. Parameter names are identical for vertex and fragment programs, although the values differ for the two types of programs. Table 13-3 shows the parameter names.

Table 13-3 Program Environment and Local Parameters

Parameter Binding	Component Usage	Description
`program.env[a]`	(x,y,z,w)	Program environment parameter `a`
`program.env[a..b]`	(x,y,z,w)	Program environment parameters `a` through `b`
`program.local[a]`	(x,y,z,w)	Program local parameter `a`
`program.local[a..b]`	(x,y,z,w)	Program local parameters a through `b`

In Table 13-3, the notation `[a]` refers to a single parameter indexed by the constant value a, and the notation `[a..b]` refers to an array of parameters indexed by the constant values `a` and `b`, which may be bound to a corresponding array using the PARAM statement. When specifying arrays, `b` must be greater than `a`, and both `a` and `b` must be within the range of supported parameter indices for that type of parameter.

OpenGL State Parameters

Most OpenGL state can be accessed in instructions either by binding state names with the PARAM naming statement or directly by use of the state name. OpenGL state falls into several different categories, which are discussed separately in the following subsections:

- Material Property Bindings
- Light Property Bindings

- Texture Coordinate Generation Property Bindings
- Texture Enviroment Property Bindings
- Fog Property Bindings
- Clip Plan Property Bindings
- Point Property Bindings
- Depth Property Bindings
- Matrix Property Bindings

Most OpenGL state categories are available to both vertex and fragment programs, but a few categories are available only to vertex programs, or only to fragment programs. OpenGL state categories restricted to one type of program are identified in their respective subsection.

Material Property Bindings

Material property bindings provide access to the OpenGL state specified with **glMaterialf()**. Table 13-4 shows the possible bindings.

Table 13-4 Material Property Bindings

Parameter Binding	Component Usage	Description
`state.material.ambient`	(r,g,b,a)	Front ambient material color
`state.material.diffuse`	(r,g,b,a)	Front diffuse material color
`state.material.specular`	(r,g,b,a)	Front specular material color
`state.material.emission`	(r,g,b,a)	Front emissive material color
`state.material.shininess`	(s,0,0,1)	Front material shininess
`state.material.front.ambient`	(r,g,b,a)	Front ambient material color
`state.material.front.diffuse`	(r,g,b,a)	Front diffuse material color
`state.material.front.specular`	(r,g,b,a)	Front specular material color
`state.material.front.emission`	(r,g,b,a)	Front emissive material color
`state.material.front.shininess`	(s,0,0,1)	Front material shininess
`state.material.back.ambient`	(r,g,b,a)	Back ambient material color

Table 13-4 Material Property Bindings **(continued)**

Parameter Binding	Component Usage	Description
`state.material.back.diffuse`	(r,g,b,a)	Back diffuse material color
`state.material.back.specular`	(r,g,b,a)	Back specular material color
`state.material.back.emission`	(r,g,b,a)	Back emissive material color
`state.material.back.shininess`	(s,0,0,1)	Back material shininess

For material shininess, the `.x` component is filled with the material's specular exponent, and the `.y`, `.z`, and `.w` components are filled with `0`, `0`, and `1`, respectively. Bindings containing `.back` refer to the back material; all other bindings refer to the front material.

Material properties can be changed between **glBegin()** and **glEnd()**, either directly by calling **glMaterialf()** or indirectly through color material. However, such property changes are not guaranteed to update parameter bindings until the following **glEnd()** command. Parameter variables bound to material properties changed between **glBegin()** and **glEnd()** are undefined until the following **glEnd()** command.

Light Property Bindings

Light property bindings provide access to the OpenGL state specified with **glLightf()** and **glLightModelf()** and to some derived properties generated from light and light model state values. Table 13-5 shows the possible light property bindings.

Table 13-5 Light Property Bindings

Parameter Binding	Component Usage	Description
`state.light[`*n*`].ambient`	(r,g,b,a)	Light *n* ambient color
`state.light[`*n*`].diffuse`	(r,g,b,a)	Light *n* diffuse color
`state.light[`*n*`].specular`	(r,g,b,a)	Light *n* specular color
`state.light[`*n*`].position`	(x,y,z,w)	Light *n* position light *n* attenuation
`state.light[`*n*`].attenuation`	(a,b,c,e)	Light *n* attenuation constants and spot light exponent

Table 13-5 Light Property Bindings **(continued)**

Parameter Binding	Component Usage	Description
`state.light[`*n*`].spot.direction`	(x,y,z,c)	Light *n* spot direction and cutoff angle cosine
`state.light[`*n*`].half`	(x,y,z,1)	Light *n* infinite half-angle
`state.lightmodel.ambient`	(r,g,b,a)	Light model ambient color
`state.lightmodel.scenecolor`	(r,g,b,a)	Light model front scene color
`state.lightmodel.front.scenecolor`	(r,g,b,a)	Light model front scene color
`state.lightmodel.back.scenecolor`	(r,g,b,a)	Light model back scene color
`state.lightprod[`*n*`].ambient`	(r,g,b,a)	Light *n* / front material ambient color product
`state.lightprod[`*n*`].diffuse`	(r,g,b,a)	Light *n* / front material diffuse color product
`state.lightprod[`*n*`].specular`	(r,g,b,a)	Light *n* / front material specular color product
`state.lightprod[`*n*`].front.ambient`	(r,g,b,a)	Light *n* / front material ambient color product
`state.lightprod[`*n*`].front.diffuse`	(r,g,b,a)	Light *n* / front material diffuse color product
`state.lightprod[`*n*`].front.specular`	(r,g,b,a)	Light *n* / front material specular color product
`state.lightprod[`*n*`].back.ambient`	(r,g,b,a)	Light *n* / back material ambient color product
`state.lightprod[`*n*`].back.diffuse`	(r,g,b,a)	Light *n* / back material diffuse color product
`state.lightprod[`*n*`].back.specular`	(r,g,b,a)	Light *n* / back material specular color product

The [*n*] syntax indicates a specific light (`GL_LIGHT`*n*).

For the following bindings, the `.x`, `.y`, `.z`, and `.w` components are filled with the red, green, blue, and alpha components, respectively, of the corresponding light color:

- `state.light[`*n*`].ambient`
- `state.light[`*n*`].diffuse`
- `state.light[`*n*`].specular`

For `state.light[`*n*`].position`, the `.x`, `.y`, `.z`, and `.w` components are filled with the X, Y, Z, and W components, respectively, of the corresponding light position.

For `state.light[`*n*`].attenuation`, the `.x`, `.y`, and `.z` components are filled with the corresponding light constant, linear, and quadratic attenuation parameters. The `.w` component is filled with the spot light exponent of the corresponding light.

For `state.light[`*n*`].spot.direction`, the `.x`, `.y`, and `.z` components variable are filled with the `.x`, `.y`, and `.z` components of the spot light direction of the corresponding light, respectively. The `.w` component is filled with the cosine of the spot light cutoff angle of the corresponding light.

For `state.light[`*n*`].half`, the `.x`, `.y`, and `.z` components of the program parameter variable are filled with the x, y, and z components, respectively, of the following normalized infinite half-angle vector:

```
h_inf = || P + (0, 0, 1) ||
```

The `.w` component of is filled with `1`. In the computation of `h_inf`, `P` consists of the X, Y, and Z coordinates of the normalized vector from the eye position to the eye-space light position. `h_inf` is defined to correspond to the normalized half-angle vector when using an infinite light (W coordinate of the position is zero) and an infinite viewer. For local lights or a local viewer, `h_inf` is well-defined but does not match the normalized half-angle vector, which will vary depending on the vertex position.

For `state.lightmodel.ambient`, the `.x`, `.y`, `.z`, and `.w` components of the program parameter variable are filled with the red, green, blue, and alpha components of the light model ambient color, respectively.

For `state.lightmodel.scenecolor` or `state.lightmodel.front.scenecolor`, the `.x`, `.y`, and `.z` components of the program parameter variable are filled with the red, green, and blue components respectively of the front scene color, defined by the following:

```
c_scene = a_cs * a_cm + e_cm
```

The operand `a_cs` is the light model ambient color, `a_cm` is the front ambient material color, and `e_cm` is the front emissive material color with computations performed separately for each color component. The `.w` component of the program parameter variable is filled with the alpha component of the front diffuse material color.

For `state.lightmodel.back.scenecolor`, a similar back scene color computed using back-facing material properties is used. The front and back scene colors match the values that would be assigned to vertices using conventional lighting if all lights were disabled.

For bindings beginning with `state.lightprod[`*n*`]`, the `.x`, `.y`, and `.z` components of the program parameter variable are filled with the red, green, and blue components, respectively, of the corresponding light product. The three light product components are the products of the corresponding color components of the specified material property and the light color of the corresponding light (see Table 13-5). The `.w` component of the program parameter variable is filled with the alpha component of the specified material property.

Light products depend on material properties, which can be changed between **glBegin**() and **glEnd()**. Such property changes are not guaranteed to take effect until the following **glEnd()** command. Program parameter variables bound to light products whose corresponding material property changes between **glBegin()** and **glEnd()** are undefined until the following **glEnd()** command.

Texture Coordinate Generation Property Bindings

Texture coordinate generation property bindings are only available within vertex programs. They provide access to the OpenGL state specified with **glTexGenf()**. Table 13-6 shows the possible texture coordinate generation property bindings.

Table 13-6 Texture Coordinate Generation Property Bindings

ParameterBinding	Component Usage	Description
`state.texgen[`*n*`].eye.s`	(a,b,c,d)	**glTexGen()** eye linear plane coefficients, s coord, unit *n*
`state.texgen[`*n*`].eye.t`	(a,b,c,d)	**glTexGen()** eye linear plane coefficients, `t` coord, unit *n*
`state.texgen[`*n*`].eye.r`	(a,b,c,d)	**glTexGen()** eye linear plane coefficients, `r` coord, unit *n*

Table 13-6 Texture Coordinate Generation Property Bindings **(continued)**

ParameterBinding	Component Usage	Description
`state.texgen[`*n*`].eye.q`	(a,b,c,d)	**glTexGen()** eye linear plane coefficients, `q` coord, unit *n*
`state.texgen[`*n*`].object.s`	(a,b,c,d)	glTexGen() object linear plane coefficients, `s` coord, unit *n*
`state.texgen[`*n*`].object.t`	(a,b,c,d)	**glTexGen()** object linear plane coefficients, `t` coord, unit *n*
`state.texgen[`*n*`].object.r`	(a,b,c,d)	**glTexGen()** object linear plane coefficients, `r` coord, unit *n*
`state.texgen[`*n*`].object.q`	(a,b,c,d)	**glTexGen()** object linear plane coefficients, `q` coord, unit *n*

The [*n*] syntax indicates a specific texture unit. If omitted, values for texture unit zero will be bound.

For the `state.texgen[`*n*`].object` bindings, the `.x`, `.y`, `.z`, and `.w` components of the parameter variable are filled with the *p1*, *p2*, *p3*, and *p4* values, respectively, specified to **glTexGen()** as the `GL_OBJECT_LINEAR` coefficients for the specified texture coordinate `.s`, `.t`, `.r`, `.q`.

For the `state.texgen[`*n*`].eye` bindings, the `.x`, `.y`, `.z`, and `.w` components of the parameter variable are filled with the *p1′*, *p2′*, *p3′*, and *p4′* values, respectively, specified to **glTexGen()** as the `GL_EYE_LINEAR` coefficients for the specified texture coordinate `.s`, `.t`, `.r`, `.q`.

Texture Environment Property Bindings

Texture environment property bindings are only available within fragment programs. They provide access to the texture environment color specified with **glTexEnvf()**. Table 13-7 shows the possible texture environment property bindings.

Table 13-7 Texture Environment Property Bindings

Parameter Binding	Component Usage	Description
`state.texenv.color`	(r,g,b,a)	Texture environment zero color
`state.texenv[`*n*`].color`	(r,g,b,a)	Texture environment *n* color

The [*n*] syntax indicates a specific texture unit. If omitted, values for texture unit zero will be bound.

For `state.texenv[`*n*`].color`, the `.x`, `.y`, `.z`, and `.w` components of the parameter variable are filled with the red, green, blue, and alpha components, respectively, of the corresponding texture environment color. Note that only legacy texture units within the range specified by `GL_MAX_TEXTURE_UNITS` have texture environment state. Texture image units and texture coordinate sets do not have associated texture environment state.

Fog Property Bindings

Fog property bindings provide access to the OpenGL state specified with **glFogf()**. Table 13-8 shows the possible fog property bindings.

Table 13-8 Fog Property Bindings

Parameter Binding	Component Usage	Description
`state.fog.color`	(r,g,b,a)	RGB fog color
`state.fog.params`	(d,s,e,r)	Fog density, linear start and end, and 1/(*end* – *start*)

For `state.fog.color`, the `.x`, `.y`, `.z`, and .w components of the parameter variable are filled with the red, green, blue, and alpha, respectively, of the fog color.

For `state.fog.params`, the `.x`, `.y`, and `.z` components of the parameter variable are filled with the fog density, linear fog start, and linear fog end parameters, respectively. The `.w` component is filled with 1 / (*end* – *start*), where *end* and *start* are the linear fog end and start parameters, respectively.

Clip Plane Property Bindings

Clip plane property bindings are only available within vertex programs. They provide access to the OpenGL state specified with **glClipPlane()**. Table 13-9 shows the possible clip plane property bindings.

Table 13-9 Clip Plane Property Bindings

Parameter Binding	Component Usage	Description
`state.clip[`*n*`].plane`	(a,b,c,d)	Clip plane *n* coefficients

The [*n*] syntax indicates a specific clip plane (`GL_CLIP_PLANE`*n*).

For `state.clip[`*n*`].plane`, the `.x`, `.y`, `.z`, and `.w` components of the parameter variable are filled with the eye-space transformed coefficients *p1′*, *p2′*, *p3′*, and *p4′*, respectively, of the corresponding clip plane.

Point Property Bindings

Point property bindings are only available within vertex programs. They provide access to the OpenGL state specified with the **glPointParameterfvARB()** command (if the ARB_point_parameters extension is supported). Table 13-10 shows the possible point property bindings.

Table 13-10 Point Property Bindings

Parameter Binding	Component Usage	Description
`state.point.size`	(s,n,x,f)	Point size, minimum and maximum size clamps, and fade threshold
`state.point.attenuation`	(a,b,c,1)	Point size attenuation constants

For `state.point.size`, the `.x`, `.y`, `.z`, and `.w` components of the parameter variable are filled with the point size, minimum point size, maximum point size, and fade threshold, respectively.

For state.point.attenuation, the `.x`, `.y`, and `.z` components of the parameter variable are filled with the constant, linear, and quadratic point size distance attenuation parameters (a, b, and c), respectively. The `.w` component is filled with `1`.

Depth Property Bindings

Depth property bindings are only available within fragment programs. They provide access to the OpenGL state specified with **glDepthRange()**. Table 13-11 shows the possible depth property bindings.

Table 13-11 Depth Property Bindings

Parameter Binding	Component Usage	Description
`state.depth.range`	(n,f,d,1)	Depth range near, far, and far – near (d)

For `state.depth.range`, the `.x` and `.y` components of the parameter variable are filled with the mappings of near and far clipping planes to window coordinates, respectively. The `.z` component is filled with the difference of the mappings of near and far clipping planes, *far – near*. The `.w` component is filled with `1`.

Matrix Property Bindings

Matrix property bindings provide access to the OpenGL state specified with commands that load and multiply matrices, such as **glMatrixMode()** and **glLoadMatrixf()**. Table 13-12 shows the possible matrix property bindings.

Table 13-12 Matrix Property Bindings

Parameter Binding	Description
`state.matrix.modelview[`*n*`]`	Modelview matrix *n*
`state.matrix.projection`	Projection matrix
`state.matrix.mvp`	Modelview projection matrix
`state.matrix.texture[`*n*`]`	Texture matrix *n*
`state.matrix.palette[`*n*`]`	Modelview palette matrix *n*
`state.matrix.program[`*n*`]`	Program matrix *n*

The [*n*] syntax indicates a specific matrix number. For `.modelview` and `.texture`, a matrix number is optional, and matrix zero will be bound if the matrix number is omitted. The field .program refers to generic program matrices, which are defined as described in section "Generic Program Matrix Specification" on page 351. The field `.palette` refers to the matrix palette defined with the ARB_matrix_palette extension;

since this extension is not currently supported on Onyx4 and Silicon Graphics Prism systems, these state values may not be bound.

The base matrix bindings may be further modified by a inverse/transpose selector and a row selector. If the beginning of a parameter binding matches any of the matrix binding names listed in Table 13-12, the binding corresponds to a 4x4 matrix (instead of a four-element vector, as is true of other parameter bindings). If the parameter binding is followed by `.inverse`, `.transpose`, or `.invtrans`, the inverse, transpose, or transpose of the inverse, respectively, of the specified matrix is selected. Otherwise, the specified matrix selected. If the specified matrix is poorly conditioned (singular or nearly so), its inverse matrix is undefined.

The binding name `state.matrix.mvp` refers to the product of modelview matrix zero and the projection matrix, as defined in the following:

`MVP =` *P* `*` *M0*

The operand *P* is the projection matrix and *M0* is the modelview matrix zero.

If the selected matrix is followed by `.row[`*a*`]`, the `.x`, `.y`, `.z`, and `.w` components of the parameter variable are filled with the four entries of row a of the selected matrix. In the following example, the variable `m0` is set to the first row (row 0) of modelview matrix 1, and `m1` is set to the last row (row 3) of the transpose of the projection matrix:

```
PARAM m0 = state.matrix.modelview[1].row[0];
PARAM m1 = state.matrix.projection.transpose.row[3];
```

For parameter array bindings, multiple rows of the selected matrix can be bound. If the selected matrix binding is followed by `.row[`*a*`..`*b*`]`, the result is equivalent to specifying matrix rows *a* through *b* in order. A program will fail to load if *a* is greater than *b*. If no row selection is specified, rows 0 through 3 are bound in order. In the following code, the array `m2` has two entries, containing rows 1 and 2 of program matrix zero, and `m3` has four entries, containing all four rows of the transpose of program matrix zero:

```
PARAM m2[] = { state.matrix.program[0].row[1..2] };
PARAM m3[] = { state.matrix.program[0].transpose };
```

Vertex and Fragment Program Output

Output used by later stages of the OpenGL pipeline can be accessed in instructions either by binding output names with the `OUTPUT` statement or directly by use of the output name. Output from vertex and fragment programs is described in this section.

Vertex Program Output

Table 13-13 lists the possible types of vertex program output. Components labelled * are unused.

Table 13-13 Vertex Program Output

Output Binding	Component Usage	Description
`result.position`	(x,y,z,w)	Position in clip coordinates
`result.color`	(r,g,b,a)	Front-facing primary color
`result.color.primary`	(r,g,b,a)	Front-facing primary color
`result.color.secondary`	(r,g,b,a)	Front-facing secondary color
`result.color.front`	(r,g,b,a)	Front-facing primary color
`result.color.front.primary`	(r,g,b,a)	Front-facing primary color
`result.color.front.secondary`	(r,g,b,a)	Front-facing secondary color
`result.color.back`	(r,g,b,a)	Back-facing primary color
`result.color.back.primary`	(r,g,b,a)	Back-facing primary color
`result.color.back.secondary`	(r,g,b,a)	Back-facing secondary color
`result.fogcoord`	(f,*,*,*)	Fog coordinate
`result.pointsize`	(s,*,*,*)	Point size
`result.texcoord`	(s,t,r,q)	Texture coordinate, unit 0
`result.texcoord[`*n*`]`	(s,t,r,q)	Texture coordinate, unit *n*

For `result.position`, updates to the `.x`, `.y`, `.z`, and `.w` components of the result variable modify the X, Y, Z, and W components, respectively, of the transformed vertex's clip coordinates. Final window coordinates are generated for the vertex based on its clip coordinates.

For bindings beginning with `result.color`, updates to the `.x`, `.y`, `.z`, and `.w` components of the result variable modify the red, green, blue, and alpha components, respectively, of the corresponding vertex color attribute. Color bindings that do not specify front or back are consided to refer to front-facing colors. Color bindings that do not specify primary or secondary are considered to refer to primary colors.

For `result.fogcoord`, updates to the `.x` component of the result variable set the transformed vertex's fog coordinate. Updates to the `.y`, `.z`, and `.w` components of the result variable have no effect.

For `result.pointsize`, updates to the `.x` component of the result variable set the transformed vertex's point size. Updates to the `.y`, `.z`, and `.w` components of the result variable have no effect.

For `result.texcoord` or `result.texcoord[`*n*`]`, updates to the `.x`, `.y`, `.z`, and `.w` components of the result variable set the `s`, `t`, `r`, and `q` components, respectively, of the transformed vertex's texture coordinates for texture unit *n*. If [*n*] is omitted, texture unit zero is selected.

All output is undefined at each vertex program invocation. Any results, or even individual components of results, that are not written during vertex program execution remain undefined.

Fragment Program Output

Table 13-14 lists the possible types of fragment program output. Components labelled * are unused.

Table 13-14 Fragment Program Output

Output Binding	Component Usage	Description
`result.color`	(r,g,b,a)	Color
`result.color[`*n*`]`	(r,g,b,a)	Color for draw buffer *n*
`result.depth`	(*,*,d,*)	Depth coordinate

For `result.color` or `result.color[`*n*`]`, updates to the `.x`, `.y`, `.z`, and `.w` components of the result variable modify the red, green, blue, and alpha components, respectively, of the fragment's output color for draw buffer *n*. If [*n*] is omitted, the output color for draw buffer zero is modified. However, note that the [*n*] notation is only supported if program option `ATI_draw_buffers` is specified and if the ATI_draw_buffers extension is supported.

If `result.color` is not both bound by the fragment program and written by some instruction of the program, the output color of the fragment program is undefined.

Each color output is clamped to the range [0,1] and converted to fixed-point before being passed on to further fixed-function processing.

For `result.depth`, updates to the `.z` component of the result variable modify the fragment's output depth value. If `result.depth` is not both bound by the fragment program and written by some instruction of the program, the interpolated depth value produced by rasterization is used as if fragment program mode is not enabled. Writes to any component of depth other than the `.z` component have no effect.

The depth output is clamped to the range [0,1] and converted to fixed-point, as if it were a window Z value before being passed on to further fixed-function processing.

Program Parameter Specification

The preceding section "Vertex and Fragment Program Parameters" on page 333 describes program parameters in terms of how they are accessed within a pipeline program. To set the value of a program parameter, call one of the following commands:

```
void glProgramLocalParameter4fARB(GLenum target, GLuint index, GLfloat x,
                                  GLfloat y, GLfloat z, GLfloat w);
void glProgramLocalParameter4fvARB(GLenum target, GLuint index,
                                   const GLfloat *params);
void glProgramLocalParameter4dARB(GLenum target, GLuint index, GLdouble x,
                                  GLdouble y, GLdouble z, GLdouble w);
void glProgramLocalParameter4dvARB(GLenum target, GLuint index,
                                   const GLdouble *params);
void glProgramEnvParameter4fARB(GLenum target, GLuint index, GLfloat x,
                                GLfloat y, GLfloat z, GLfloat w);
void glProgramEnvParameter4fvARB(GLenum target, GLuint index,
                                 const GLfloat *params);
void glProgramEnvParameter4dARB(GLenum target, GLuint index, GLdouble x,
                                GLdouble y, GLdouble z, GLdouble w);
void glProgramEnvParameter4dvARB(GLenum target, GLuint index,
                                 const GLdouble *params);
```

The **glProgramLocal*()** commands update the value of the program local parameter numbered index belonging to the program currently bound to *target*, and the **glProgramEnv*()** commands update the value of the program environment parameter numbered index for *target*. The argument *target* may be either `GL_VERTEX_PROGRAM_ARB` or `GL_FRAGMENT_PROGRAM_ARB`.

The scalar forms of the commands set the first, second, third, and fourth components of the specified parameter to the passed `x`, `y`, `z` and `w` values. The vector forms of the commands set the values of the specified parameter to the four values pointed to by *params*.

To query the value of a program local parameter, call one of the following commands:

```
void glGetProgramLocalParameterfvARB(GLenum target, GLuint index,
                                     GLfloat *params);
void glGetProgramLocalParameterdvARB(GLenum target, GLuint index,
                                     GLdouble *params);
```

To query the value of a program environment parameter, call one of the following commands:

```
void glGetProgramEnvParameterfvARB(GLenum target, GLuint index,
                                   GLfloat *params);
void glGetProgramEnvParameterdvARB(GLenum target, GLuint index,
                                   GLdouble *params);
```

For both local and environment parameters, the four components of the specified parameter are copied to the target array *params*.

The number of program local and environment parameters supported for each target type may be queried as described in section "Program Resource Limits and Usage" on page 372.

Generic Vertex Attribute Specification

The section "Vertex and Fragment Attributes" on page 329 describes vertex attributes in terms of how they are accessed within a vertex program. This section lists the commands for specifying vertex attributes and also describes attribute aliasing.

Commands

To set the value of a vertex attribute, call one of the following commands:

```
void glVertexAttrib1sARB(GLuint index, GLshort x);
void glVertexAttrib1fARB(GLuint index, GLfloat x);
void glVertexAttrib1dARB(GLuint index, GLdouble x);
void glVertexAttrib2sARB(GLuint index, GLshort x, GLshort y);
void glVertexAttrib2fARB(GLuint index, GLfloat x, GLfloat y);
```

```
void glVertexAttrib2dARB(GLuint index, GLdouble x, GLdouble y);
void glVertexAttrib3sARB(GLuint index, GLshort x, GLshort y, GLshort z);
void glVertexAttrib3fARB(GLuint index, GLfloat x, GLfloat y, GLfloat z);
void glVertexAttrib3dARB(GLuint index, GLdouble x, GLdouble y,
                         GLdouble z);
void glVertexAttrib4sARB(GLuint index, GLshort x, GLshort y, GLshort z,
                         GLshort w);
void glVertexAttrib4fARB(GLuint index, GLfloat x, GLfloat y, GLfloat z,
                         GLfloat w);
void glVertexAttrib4dARB(GLuint index, GLdouble x, GLdouble y,
                         GLdouble z, GLdouble w);
void glVertexAttrib4NubARB(GLuint index, GLubyte x, GLubyte y,
                           GLubyte z, GLubyte w);
void glVertexAttrib1svARB(GLuint index, const GLshort *v);
void glVertexAttrib1fvARB(GLuint index, const GLfloat *v);
void glVertexAttrib1dvARB(GLuint index, const GLdouble *v);
void glVertexAttrib2svARB(GLuint index, const GLshort *v);
void glVertexAttrib2fvARB(GLuint index, const GLfloat *v);
void glVertexAttrib2dvARB(GLuint index, const GLdouble *v);
void glVertexAttrib3svARB(GLuint index, const GLshort *v);
void glVertexAttrib3fvARB(GLuint index, const GLfloat *v);
void glVertexAttrib3dvARB(GLuint index, const GLdouble *v);
void glVertexAttrib4bvARB(GLuint index, const GLbyte *v);
void glVertexAttrib4svARB(GLuint index, const GLshort *v);
void glVertexAttrib4ivARB(GLuint index, const GLint *v);
void glVertexAttrib4ubvARB(GLuint index, const GLubyte *v);
void glVertexAttrib4usvARB(GLuint index, const GLushort *v);
void glVertexAttrib4uivARB(GLuint index, const GLuint *v);
void glVertexAttrib4fvARB(GLuint index, const GLfloat *v);
void glVertexAttrib4dvARB(GLuint index, const GLdouble *v);
void glVertexAttrib4NbvARB(GLuint index, const GLbyte *v);
void glVertexAttrib4NsvARB(GLuint index, const GLshort *v);
void glVertexAttrib4NivARB(GLuint index, const GLint *v);
void glVertexAttrib4NubvARB(GLuint index, const GLubyte *v);
void glVertexAttrib4NusvARB(GLuint index, const GLushort *v);
void glVertexAttrib4NuivARB(GLuint index, const GLuint *v);
```

These commands update the value of the generic vertex attribute numbered *index*. The scalar forms set the first, second, third, and fourth components of the specified attribute to the passed `x`, `y`, `z` and `w` values, and the vector forms set the values of the specified attribute to the values pointed to by *v*.

If fewer than four values are passed (for the **glVertexAttrib1*()**, **glVertexAttrib2*()**, and **glVertexAttrib3*()** forms of the commands), unspecified values of `y` and `z` default to `0.0`, and unspecified values of `w` default to `1.0`.

The **glVertexAttrib4N*()** forms of the commands specify attributes with fixed-point coordinates. The specified fixed-point values are scaled to the range [0,1] (for unsigned forms of the commands) of to the range [–1,1] (for signed forms of the commands) in the same fashion as for the **glNormal*()** commands.

The number of vertex attributes supported for each target type may be queried as described in section "Program Resource Limits and Usage" on page 372.

Attribute Aliasing

Setting generic vertex attribute 0 specifies a vertex; the four vertex coordinates are taken from the values of attribute 0. A **glVertex*()** command is completely equivalent to the corresponding **glVertexAttrib()** command with an index of zero. Setting any other generic vertex attribute updates the current values of the attribute. There are no current values for vertex attribute 0.

Implementations may, but do not necessarily, use the same storage for the current values of generic and certain conventional vertex attributes. When any generic vertex attribute other than 0 is specified, the current values for the corresponding conventional attribute aliased with that generic attribute, as described in the Table 13-1, become undefined. Similarly, when a conventional vertex attribute is specified, the current values for the corresponding generic vertex attribute become undefined. For example, setting the current normal will leave generic vertex attribute 2 undefined, and conversely so.

Generic vertex attributes may also be specified when drawing by using vertex arrays. An array of per-vertex attribute values is defined by making the following call:

```
void glVertexAttribPointerARB(GLuint index, GLint size, GLenum type,
              GLboolean normalized, GLsizei stride, const GLvoid *pointer);
```

The arguments are defined as follows:

size	Specifies the number of elements per attribute and must be 1, 2, 3, or 4.
type	Specifies the type of data in the array and must be one of `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_FLOAT`, or `GL_DOUBLE`.

stride, pointer Specifies the offset in basic machine units from one attribute value to the next in the array starting at *pointer*. As with other vertex array specification calls, a stride of zero indicates that fog coordinates are tightly packed in the array.

normalized Specifies if fixed-point values will be normalized. If *normalized* is `GL_TRUE`, fixed-point values will be normalized (in the same fashion as the **glVertexAttrib4N*()** commands just described). Otherwise, fixed-point values are used unchanged.

To enable or disable generic vertex attributes when drawing vertex arrays, call one of the following commands:

```
void glEnableVertexAttribArrayARB(GLuint index);
void glDisableVertexAttribArrayARB(GLuint index);
```

The number of program local and environment parameters supported for each target type may be queried as described in section "Program Resource Limits and Usage" on page 372.

Generic Program Matrix Specification

Programs may use additional matrices, referred to as *generic program matrices*, from the OpenGL state. These matrices are specified using the same commands—for example, **glLoadMatrixf()**—as for other matrices such as modelview and projection. To set the current OpenGL matrix mode to operate on generic matrix *n*, call **glMatrixMode()** with a mode argument of `GL_MATRIX0_ARB` + *n*.

The number of program matrices supported may be queried as described in section "Program Resource Limits and Usage" on page 372.

Program Instruction Summary

The tables in this section summarize the complete instruction set supported for pipeline programs. In the Input and Output columns, the tables use the following notation:

v Indicates a floating-point vector input or output.

s Indicates a floating-point scalar input.

ssss	Indicates a scalar output replicated across a four-component result vector.
a	Indicates a single address register component.

Note: As described in section "Program Instructions" on page 326, most fragment program instructions support clamping when writing instruction output by appending the suffix _SAT to the instruction mnemonic.

Table 13-15 summarizes instructions supported in both fragment and vertex programs.

Table 13-15 Program Instructions (Fragment and Vertex Programs)

Instruction	Input	Input	Output	Description
ABS		v	v	Absolute value
ADD		v,v	v	Add
DP3		v,v	ssss	Three-component dot product
DP4		v,v	ssss	Four-component dot product
DPH		v,v	ssss	Homogeneous dot product
DST		v,v	v	Distance vector
EX2		s	ssss	Exponentiate with base 2
FLR		v	v	Floor
FRC		v	v	Fraction
LG2		s	ssss	Logarithm base 2
LIT		v	v	Compute light coefficients
MAD		v,v,v	v	Multiply and add
MAX		v,v	v	Maximum
MIN		v,v	v	Minimum
MOV		v	v	Move
MUL		v,v	v	Multiply

Table 13-15 Program Instructions (Fragment and Vertex Programs) **(continued)**

Instruction	Input	Input	Output	Description
POW		s,s	ssss	Exponentiate
RCP		s	ssss	Reciprocal
RSQ		s	ssss	Reciprocal square root
SGE		v,v	v	Set on greater than or equal
SLT		v,v	v	Set on less than
SUB		v,v	v	Subtract
SWZ		v	v	Extended swizzle
XPD		v,v	v	Cross product

Table 13-16 summarizes instructions supported only in fragment programs.

Table 13-16 Program Instructions (Fragment Programs Only)

Instruction	Input	Output	Description
CMP	v,v,v	v	Compare
COS	s	ssss	Cosine with reduction to [–pi,pi]
KIL	v	v	Kill fragment
LRP	v,v,v	v	Linear interpolation
SCS	s	ss--	Sine/cosine without reduction
SIN	s	ssss	Sine with reduction to [–pi,pi]
TEX	v,u,t	v	Texture sample
TXB	v,u,t	v	Texture sample with bias
TXP	v,u,t	v	Texture sample with projection

Table 13-17 summarizes instructions supported only in vertex programs.

Table 13-17 Program Instructions (Vertex Programs Only)

Instruction	Input	Output	Description
ARL	s	a	Address register load
EXP	s	v	Exponential base 2 (approximate)
LOG	s	v	Logarithm base 2 (approximate)

The following subsections describe each instruction in detail:

- "Fragment and Vertex Program Instructions"
- "Fragment Program Instructions"
- "Vertex Program Instructions"

As shown in the preceding tables, most instructions are supported in both vertex and fragment programs.

Each subsection contains pseudo code describing the instruction. Instructions will have up to three operands, referred to as `op0`, `op1`, and `op2`.

Operands are loaded according to the component selection and modification rules. For a vector operand, these rules are referred to as the **VectorLoad()** operation. For a scalar operand, they are referred to as the **ScalarLoad()** operation.

The variables `tmp`, `tmp0`, `tmp1`, and `tmp2` describe scalars or vectors used to hold intermediate results in the instruction.

Most instructions will generate a result vector called `result`. The result vector is then written to the destination register specified in the instruction possibly with destination write masking and, if the `_SAT` form of the instruction is used, with destination clamping, as described previously.

Fragment and Vertex Program Instructions

The instructions described here are supported in both fragment and vertex programs.

ABS—Absolute Value

The ABS instruction performs a component-wise absolute value operation on the single operand to yield a result vector.

Pseudo code:

```
tmp = VectorLoad(op0);
result.x = fabs(tmp.x);
result.y = fabs(tmp.y);
result.z = fabs(tmp.z);
result.w = fabs(tmp.w);
```

ADD—Add

The ADD instruction performs a component-wise add of the two operands to yield a result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x + tmp1.x;
result.y = tmp0.y + tmp1.y;
result.z = tmp0.z + tmp1.z;
result.w = tmp0.w + tmp1.w;
```

The following rules apply to addition:

1. x + y == y + x, for all x and y
2. x + 0.0 == x, for all x

DP3—Three-Component Dot Product

The DP3 instruction computes a three-component dot product of the two operands (using the first three components) and replicates the dot product to all four components of the result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + (tmp0.z * tmp1.z);
result.x = dot;
result.y = dot;
result.z = dot;
result.w = dot;
```

DP4—Four-Component Dot Product

The `DP4` instruction computes a four-component dot product of the two operands and replicates the dot product to all four components of the result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1):
dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + (tmp0.z * tmp1.z) +
(tmp0.w * tmp1.w);
result.x = dot;
result.y = dot;
result.z = dot;
result.w = dot;
```

`DPH`—Homogeneous Dot Product

The `DPH` instruction computes a three-component dot product of the two operands (using the `x`, `y`, and `z` components), adds the `w` component of the second operand, and replicates the sum to all four components of the result vector. This is equivalent to a four-component dot product where the `w` component of the first operand is forced to `1.0`.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1):
dot = (tmp0.x * tmp1.x) + (tmp0.y * tmp1.y) + (tmp0.z * tmp1.z) +
tmp1.w;
result.x = dot;
result.y = dot;
result.z = dot;
result.w = dot;
```

DST—Distance Vector

The DST instruction computes a distance vector from two specially formatted operands. The first operand should be of the form [*NA*, *d*^2, *d*^2, *NA*] and the second operand should be of the form [*NA*, 1/*d*, *NA*, 1/*d*], where *NA* values are not relevant to the calculation and *d* is a vector length. If both vectors satisfy these conditions, the result vector will be of the form [1.0, *d*, *d*^2, 1/*d*].

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = 1.0;
result.y = tmp0.y * tmp1.y;
result.z = tmp0.z;
result.w = tmp1.w;
```

Given an arbitrary vector, *d*^2 can be obtained using the DP3 instruction (using the same vector for both operands) and 1/*d* can be obtained from *d*^2 using the RSQ instruction.

This distance vector is useful for light attenuation calculations: a DP3 operation using the distance vector and an attenuation constant vector as operands will yield the attenuation factor.

EX2—Exponentiate with Base 2

The EX2 instruction approximates 2 raised to the power of the scalar operand and replicates the approximation to all four components of the result vector.

Pseudo code:

```
tmp = ScalarLoad(op0);
result.x = Approx2ToX(tmp);
result.y = Approx2ToX(tmp);
result.z = Approx2ToX(tmp);
result.w = Approx2ToX(tmp);
```

FLR—Floor

The FLR instruction performs a component-wise floor operation on the operand to generate a result vector. The floor of a value is defined as the largest integer less than or equal to the value. The floor of 2.3 is 2.0; the floor of –3.6 is –4.0.

Pseudo code:

```
tmp = VectorLoad(op0);
result.x = floor(tmp.x);
result.y = floor(tmp.y);
result.z = floor(tmp.z);
result.w = floor(tmp.w);
```

FRC—Fraction

The FRC instruction extracts the fractional portion of each component of the operand to generate a result vector. The fractional portion of a component is defined as the result after subtracting off the floor of the component (see the FLR instruction) and is always in the range [0.0, 1.0].

For negative values, the fractional portion is **not** the number written to the right of the decimal point. The fractional portion of –1.7 is not 0.7; it is 0.3. The value 0.3 is produced by subtracting the floor of –1.7, which is –2.0, from –1.7.

Pseudo code:

```
tmp = VectorLoad(op0);
result.x = fraction(tmp.x);
result.y = fraction(tmp.y);
result.z = fraction(tmp.z);
result.w = fraction(tmp.w);
```

LG2—Logarithm Base 2

The LG2 instruction approximates the base 2 logarithm of the scalar operand and replicates it to all four components of the result vector.

Pseudo code:

```
tmp = ScalarLoad(op0);
result.x = ApproxLog2(tmp);
result.y = ApproxLog2(tmp);
result.z = ApproxLog2(tmp);
result.w = ApproxLog2(tmp);
```

If the scalar operand is zero or negative, the result is undefined.

LIT—Light Coefficients

The `LIT` instruction accelerates lighting by computing lighting coefficients for ambient, diffuse, and specular light contributions. The `.x` component of the single operand is assumed to hold a diffuse dot product (such as a vertex normal dotted with the unit direction vector from the point being lit to the light position). The `.y` component of the operand is assumed to hold a specular dot product (such as a vertex normal dotted with the half-angle vector from the point being lit). The `.w` component of the operand is assumed to hold the specular exponent of the material and is clamped to the range [–128,+128] exclusive.

The `.x` component of the result vector receives the value that should be multiplied by the ambient light/material product (always `1.0`). The `.y` component of the result vector receives the value that should be multiplied by the diffuse light/material product (for example, `state.lightprod[`*n*`].diffuse`). The `.z` component of the result vector receives the value that should be multiplied by the specular light/material product (for example, `state.lightprod[`*n*`].specular`). The `.w` component of the result is the constant `1.0`.

Negative diffuse and specular dot products are clamped to `0.0`, as is done in the fixed-function per-vertex lighting operations. In addition, if the diffuse dot product is zero or negative, the specular coefficient is forced to zero.

Pseudo code:

```
tmp = VectorLoad(op0);
if (tmp.x < 0) tmp.x = 0;
if (tmp.y < 0) tmp.y = 0;
if (tmp.w < -(128.0-epsilon)) tmp.w = -(128.0-epsilon);
else if (tmp.w > 128-epsilon) tmp.w = 128-epsilon;
result.x = 1.0;
result.y = tmp.x;
result.z = (tmp.x > 0) ? ApproxPower(tmp.y, tmp.w) : 0.0;
result.w = 1.0;
```

The power approximation function **ApproxPower()** may be defined in terms of the base 2 exponentiation and logarithm approximation operations. When executed in fragment programs, the definition should be as follows:

```
ApproxPower(a,b) = Approx2ToX(b * ApproxLog2(a))
```

The functions **Approx2ToX()** and **ApproxLog2()** are as defined by the EX2 and LG2 instructions. When executed in vertex programs, the definition should be as follows:

```
ApproxPower(a,b) = RoughApprox2ToX(b * RoughApproxLog2(a))
```

The functions **RoughApprox2ToX()** and **RoughApproxLog2()** are as defined by the EXP and LOG instructions. The approximation may not be any more accurate than the underlying exponential and logarithm approximations.

Since 0^0 is defined to be 1, **ApproxPower(**0.0, 0.0**)** will produce 1.0.

MAD—Multiply and Add

The MAD instruction performs a component-wise multiply of the first two operands, and then does a component-wise add of the product to the third operand to yield a result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = tmp0.x * tmp1.x + tmp2.x;
result.y = tmp0.y * tmp1.y + tmp2.y;
result.z = tmp0.z * tmp1.z + tmp2.z;
result.w = tmp0.w * tmp1.w + tmp2.w;
```

The multiplication and addition operations in this instruction are subject to the same rules as described for the MUL and ADD instructions.

MAX—Maximum

The MAX instruction computes component-wise maximums of the values in the two operands to yield a result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x > tmp1.x) ? tmp0.x : tmp1.x;
result.y = (tmp0.y > tmp1.y) ? tmp0.y : tmp1.y;
result.z = (tmp0.z > tmp1.z) ? tmp0.z : tmp1.z;
result.w = (tmp0.w > tmp1.w) ? tmp0.w : tmp1.w;
```

MIN—Minimum

The `MIN` instruction computes component-wise minimums of the values in the two operands to yield a result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x > tmp1.x) ? tmp1.x : tmp0.x;
result.y = (tmp0.y > tmp1.y) ? tmp1.y : tmp0.y;
result.z = (tmp0.z > tmp1.z) ? tmp1.z : tmp0.z;
result.w = (tmp0.w > tmp1.w) ? tmp1.w : tmp0.w;
```

MOV—Move

The `MOV` instruction copies the value of the operand to yield a result vector.

Pseudo code:

```
result = VectorLoad(op0);
```

MUL—Multiply

The `MUL` instruction performs a component-wise multiply of the two operands to yield a result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x * tmp1.x;
result.y = tmp0.y * tmp1.y;
result.z = tmp0.z * tmp1.z;
result.w = tmp0.w * tmp1.w;
```

The following rules apply to multiplication:

1. `x` * `y` == `y` * `x`, for all `x` and `y`
2. +/-0.0 * `x` = +/-0.0 at least for all `x` that correspond to representable numbers (The IEEE *non-number* and *infinity* encodings may be exceptions.)
3. +1.0 * `x` = `x`, for all `x`

Multiplication by zero and one should be invariant, as it may be used to evaluate conditional expressions without branching.

POW—Exponentiate

The POW instruction approximates the value of the first scalar operand raised to the power of the second scalar operand and replicates it to all four components of the result vector.

Pseudo code:

```
tmp0 = ScalarLoad(op0);
tmp1 = ScalarLoad(op1);
result.x = ApproxPower(tmp0, tmp1);
result.y = ApproxPower(tmp0, tmp1);
result.z = ApproxPower(tmp0, tmp1);
result.w = ApproxPower(tmp0, tmp1);
```

The power approximation function may be implemented using the base 2 exponentiation and logarithm approximation operations in the EX2 and LG2 instructions, as shown in the following:

```
ApproxPower(a,b) = ApproxExp2(b * ApproxLog2(a))
```

Note that a logarithm may be involved even for cases where the exponent is an integer. This means that it may not be possible to exponentiate correctly with a negative base. In constrast, it is possible in a normal mathematical formulation to raise negative numbers to integer powers (for example, (–3)^2 == 9, and (–0.5)^-2 == 4).

RCP—Reciprocal

The RCP instruction approximates the reciprocal of the scalar operand and replicates it to all four components of the result vector.

Pseudo code:

```
tmp = ScalarLoad(op0);
result.x = ApproxReciprocal(tmp);
result.y = ApproxReciprocal(tmp);
result.z = ApproxReciprocal(tmp);
result.w = ApproxReciprocal(tmp);
```

The following rule applies to reciprocation:

```
ApproxReciprocal(+1.0) = +1.0
```

RSQ—Reciprocal Square Root

The RSQ instruction approximates the reciprocal of the square root of the absolute value of the scalar operand and replicates it to all four components of the result vector.

Pseudo code:

```
tmp = fabs(ScalarLoad(op0));
result.x = ApproxRSQRT(tmp);
result.y = ApproxRSQRT(tmp);
result.z = ApproxRSQRT(tmp);
result.w = ApproxRSQRT(tmp);
```

SGE—Set on Greater or Equal Than

The SGE instruction performs a component-wise comparison of the two operands. Each component of the result vector is `1.0` if the corresponding component of the first operands is greater than or equal that of the second and 0.0, otherwise.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x >= tmp1.x) ? 1.0 : 0.0;
result.y = (tmp0.y >= tmp1.y) ? 1.0 : 0.0;
result.z = (tmp0.z >= tmp1.z) ? 1.0 : 0.0;
result.w = (tmp0.w >= tmp1.w) ? 1.0 : 0.0;
```

SLT—Set on Less Than

The SLT instruction performs a component-wise comparison of the two operands. Each component of the result vector is `1.0` if the corresponding component of the first operand is less than that of the second and `0.0`, otherwise.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = (tmp0.x < tmp1.x) ? 1.0 : 0.0;
result.y = (tmp0.y < tmp1.y) ? 1.0 : 0.0;
result.z = (tmp0.z < tmp1.z) ? 1.0 : 0.0;
```

```
result.w = (tmp0.w < tmp1.w) ? 1.0 : 0.0;
```

SUB—Subtract

The SUB instruction performs a component-wise subtraction of the second operand from the first to yield a result vector.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.x - tmp1.x;
result.y = tmp0.y - tmp1.y;
result.z = tmp0.z - tmp1.z;
result.w = tmp0.w - tmp1.w;
```

SWZ—Extended Swizzle

The SWZ instruction loads the single vector operand and performs a swizzle operation more powerful than that provided for loading normal vector operands to yield an instruction vector.

The extended swizzle is expressed as the following:

SWZ result, op0, *xswz*, *yswz*, *zswz*, *wswz*

The arguments *xswz*, *yswz*, *zswz*, and *wswz* are each one of the following extended swizzle selectors:

0, +0, -0, 1, +1, -1, x, +x, -x, y, +y, -y, z, +z, -z, w, +w, or -w

For the numeric extended swizzle selectors, the result components corresponding to *xswz*, *yswz*, *zswz*, and *wswz* are loaded with the specified number. For the non-numeric extended swizzle selectors, the result components are loaded with the source component of op0 specified by the extended swizzle selector and are negated if the selector begins with the – sign.

In fragment programs, but not in vertex programs, the following extended swizzle selectors may also be used:

r, +r, -r, g, +g, -g, b, +b, -b, a, +a, or -a

Since the SWZ instruction allows for component selection and negation for each individual component, the grammar does not allow the use of the normal swizzle and negation operations allowed for vector operands in other instructions.

The following example of SWZ shows most of the possible types of selectors:

```
PARAM   color = { -0.1, 0.7, 1.2, 1.0 };
TEMP    result;

SWZ     result, color, -1, 0, x, -y;
# result now contains { -1.0, 0.0, 0.1, -0.7 }
```

XPD—Cross Product

The XPD instruction computes the cross product using the first three components of its two vector operands to generate the X, Y, and Z components of the result vector. The W component of the result vector is undefined.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
result.x = tmp0.y * tmp1.z - tmp0.z * tmp1.y;
result.y = tmp0.z * tmp1.x - tmp0.x * tmp1.z;
result.z = tmp0.x * tmp1.y - tmp0.y * tmp1.x;
```

Fragment Program Instructions

The instructions supported only in fragment programs are of two types:

- Math instructions
- Texture instructions

Math Instructions

The math instructions include the following mathematical operations common in per-pixel shading algorithms:

- CMP
- COS
- LRP
- SCS

- SIN

CMP—Compare

The CMP instructions perform a component-wise comparison of the first operand against zero and copies the values of the second or third operands based on the results of the compare.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = (tmp0.x < 0.0) ? tmp1.x : tmp2.x;
result.y = (tmp0.y < 0.0) ? tmp1.y : tmp2.y;
result.z = (tmp0.z < 0.0) ? tmp1.z : tmp2.z;
result.w = (tmp0.w < 0.0) ? tmp1.w : tmp2.w;
```

COS—Cosine

The COS instruction approximates the trigonometric cosine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [–pi,pi].

Pseudo code:

```
tmp = ScalarLoad(op0);
result.x = ApproxCosine(tmp);
result.y = ApproxCosine(tmp);
result.z = ApproxCosine(tmp);
result.w = ApproxCosine(tmp);
```

LRP—Linear Interpolation

The LRP instruction performs a component-wise linear interpolation between the second and third operands using the first operand as the blend factor.

Pseudo code:

```
tmp0 = VectorLoad(op0);
tmp1 = VectorLoad(op1);
tmp2 = VectorLoad(op2);
result.x = tmp0.x * tmp1.x + (1 - tmp0.x) * tmp2.x;
result.y = tmp0.y * tmp1.y + (1 - tmp0.y) * tmp2.y;
```

```
result.z = tmp0.z * tmp1.z + (1 - tmp0.z) * tmp2.z;
result.w = tmp0.w * tmp1.w + (1 - tmp0.w) * tmp2.w;
```

SCS—Sine/Cosine

The SCS instruction approximates the trigonometric sine and cosine of the angle specified by the scalar operand and places the cosine in the x component and the sine in the y component of the result vector. The z and w components of the result vector are undefined. The angle is specified in radians and must be in the range [–pi,pi].

Pseudo code:

```
tmp = ScalarLoad(op0);
result.x = ApproxCosine(tmp);
result.y = ApproxSine(tmp);
```

If the scalar operand is not in the range [–pi,pi], the result vector is undefined.

SIN—Sine

The SIN instruction approximates the trigonometric sine of the angle specified by the scalar operand and replicates it to all four components of the result vector. The angle is specified in radians and does not have to be in the range [–pi,pi].

Pseudo code:

```
tmp = ScalarLoad(op0);
result.x = ApproxSine(tmp);
result.y = ApproxSine(tmp);
result.z = ApproxSine(tmp);
result.w = ApproxSine(tmp);
```

Texture Instructions

The following texture instructions include texture map lookup operations and a kill instruction:

- TEX
- TXP
- TXB
- KIL

The TEX, TXP, and TXB instructions specify the mapping of 4-tuple vectors to colors of an image. The sampling of the texture works in the same fashion as the fixed-function OpenGL pipeline, except that texture environments and texture functions are not applied to the result and the texture enable hierarchy is replaced by explicit references to the desired texture target—1D, 2D, 3D, CUBE (for cubemap targets) and RECT (for texture rectangle targets, if the EXT_texture_rectangle extension is supported). These texture instructions specify how the 4-tuple is mapped into the coordinates used for sampling. The following function is used to describe the texture sampling in the descriptions below:

```
vec4 TextureSample(float s, float t, float r, float lodBias,
                   int texImageUnit, enum texTarget);
```

Note that not all three texture coordinates *s*, *t*, and *r* are used by all texture targets. In particular, 1D texture targets only use the s component, and 2D and RECT (non-power-of-two) texture targets only use the s and t components. The following descriptions of the texture instructions supply all three components, as would be the case with 3D or CUBE targets.

If a fragment program samples from a texture target on a texture image unit where the bound texture object is not complete, the result will be the vector (R, G, B, A) = (0, 0, 0, 1).

A fragment program will fail to load if it attempts to sample from multiple texture targets on the same texture image unit. For example, the following program would fail to load:

```
!!ARBfp1.0
TEX result.color, fragment.texcoord[0], texture[0], 2D;
TEX result.depth, fragment.texcoord[1], texture[0], 3D;
END
```

The KIL instruction does not sample from a texture but rather prevents further processing of the current fragment if any component of its 4-tuple vector is less than zero.

Texture Indirections

A dependent texture instruction is one that samples using a texture coordinate residing in a temporary rather than in an attribute or a parameter. A program may have a chain of dependent texture instructions, where the result of the first texture instruction is used as the coordinate for a second texture instruction, which is, in turn, used as the coordinate for a third texture instruction, etc. Each node in this chain is termed an indirection and can be thought of as a set of texture samples that execute in parallel and are followed by a sequence of ALU instructions.

Some implementations may have limitations on how long the dependency chain may be. Therefore, indirections are counted as a resource just like instructions or temporaries are counted. All programs have at least one indirection (one node in this chain) even if the program performs no texture operation. Each instruction encountered is included in this node until the program encounters a texture instruction with one of the following properties:

- Its texture coordinate is a temporary that has been previously written in the current node.
- Its result vector is a temporary that is also the operand or result vector of a previous instruction in the current node.

A new node is then started that includes the texture instruction and all subsequent instructions, and the process repeats for all instructions in the program. Note that for simplicity in counting, result writemasks and operand suffixes are not taken into consideration when counting indirections.

TEX—Map Coordinate to Color

The `TEX` instruction takes the first three components of its source vector and maps them to `s`, `t`, and `r`. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters. The resulting sample is mapped to RGBA and written to the result vector.

Pseudo code:

```
tmp = VectorLoad(op0);
result = TextureSample(tmp.x, tmp.y, tmp.z, 0.0, op1, op2);
```

TXP—Project Coordinate and Map to Color

The `TXP` instruction divides the first three components of its source vector by the fourth component and maps the results to `s`, `t`, and `r`. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters. The resulting sample is mapped to RGBA and written to the result vector. If the value of the fourth component of the source vector is less than or equal to zero, the result vector is undefined.

Pseudo code:

```
tmp = VectorLoad(op0);
tmp.x = tmp.x / tmp.w;
tmp.y = tmp.y / tmp.w;
```

```
tmp.z = tmp.z / tmp.w;
result = TextureSample(tmp.x, tmp.y, tmp.z, 0.0, op1, op2);
```

TXB—Map Coordinate to Color While Biasing Its Level Of Detail

The TXB instruction takes the first three components of its source vector and maps them to s, t, and r. These coordinates are used to sample from the specified texture target on the specified texture image unit in a manner consistent with its parameters. Additionally, before determining the mipmap level(s) to sample, the fourth component of the source vector is added with bias factors for the per-texture-object and per-texture-unit level of detail. The resulting sample is mapped to RGBA and written to the result vector.

Pseudo code:

```
tmp = VectorLoad(op0);
result = TextureSample(tmp.x, tmp.y, tmp.z, tmp.w, op1, op2);
```

KIL—Kill Fragment

Rather than mapping a coordinate set to a color, the KIL operation prevents a fragment from receiving any future processing. If any component of its source vector is negative, the processing of this fragment will be discontinued and no further output to this fragment will occur. Subsequent stages of the GL pipeline will be skipped for this fragment.

Pseudo code:

```
tmp = VectorLoad(op0);
if ((tmp.x < 0) || (tmp.y < 0) || (tmp.z < 0) || (tmp.w < 0)) {
    exit;
}
```

Vertex Program Instructions

The instructions described in this section are only supported by vertex programs. They include address register loads (ARL) as well as lower-precision (and higher-performance) exponential and logarithmic computations (EXP and LOG).

ARL—Address Register Load

The ARL instruction loads a single scalar operand and performs a floor operation to generate a signed integer scalar result.

Pseudo code:

```
result = floor(ScalarLoad(op0));
```

EXP—Exponentiate with Base 2 (approximate)

The EXP instruction computes a rough approximation of 2 raised to the power of the scalar operand. The approximation is returned in the `.z` component of the result vector. A vertex program can also use the `.x` and `.y` components of the result vector to generate a more accurate approximation by evaluating `result.x` * **f(**`result.y`**)**, where **f(***x***)** is a user-defined function that approximates 2^*x* over the domain [0.0, 1.0]. The `.w` component of the result vector is always `1.0`.

Pseudo code:

```
tmp = ScalarLoad(op0);
result.x = 2^floor(tmp);
result.y = tmp - floor(tmp);
result.z = RoughApprox2ToX(tmp);
result.w = 1.0;
```

The approximation function is accurate to at least 10 bits.

LOG—Logarithm Base 2 (approximate)

The LOG instruction computes a rough approximation of the base 2 logarithm of the absolute value of the scalar operand. The approximation is returned in the `.z` component of the result vector. A vertex program can also use the `.x` and `.y` components of the result vector to generate a more accurate approximation by evaluating `result.x` + **f(**`result.y`**)**, where **f(***x***)** is a user-defined function that approximates 2^*x* over the domain [0.0, 1.0]. The `.w` component of the result vector is always `1.0`.

Pseudo code:

```
tmp = fabs(ScalarLoad(op0));
result.x = floor(log2(tmp));
result.y = tmp / 2^(floor(log2(tmp)));
result.z = RoughApproxLog2(tmp);
result.w = 1.0;
```

The **floor(log2(**`tmp`**))** refers to the floor of the exact logarithm, which can be easily computed for standard floating point representations. The approximation function is accurate to at least 10 bits.

Program Resource Limits and Usage

You can query resources allocated and consumed by programs by making the following call:

```
void glGetProgramivARB(GLenum target, GLenum pname, GLint *params);
```

The argument target may be either GL_VERTEX_PROGRAM_ARB or GL_FRAGMENT_PROGRAM_ARB.

To determine the maximum possible resource limits for a program of the specified target type, use one of the values in Table 13-18 for *pname*. There are two types of limits:

Native limits — If native limits are not exceeded by a program, it is guaranteed that the program can execute in the graphics hardware. The parameter names for native limits are of the form GL_MAX_PROGRAM_NATIVE*.

Overall limits — If the overall limits are not exceeded, the program will execute, but possibly on a software fallback path with greatly reduced performance. The parameter names for overall limits are of the form GL_MAX_PROGRAM*.

The concepts of texture instructions and texture indirections are described in section "Fragment Program Instructions" on page 365. In a fragment program, ALU instructions are all instructions other than the texture instructions TEX, TXP, TXB, and KIL.

Table 13-18 Program Resource Limits

Resource Limit Name (overall, native)	Min Value for Vertex Programs	Min Value for Fragment Programs	Description
GL_MAX_PROGRAM_INSTRUCTIONS_ARB, GL_MAX_PROGRAM_NATIVE_INSTRUCTIONS_ARB	128	72	Maximum number of instructions declared
GL_MAX_PROGRAM_ATTRIBS_ARB, GL_MAX_PROGRAM_NATIVE_ATTRIBS_ARB	16	10	Maximum number of attributes declared

Table 13-18 Program Resource Limits **(continued)**

Resource Limit Name (overall, native)	Min Value for Vertex Programs	Min Value for Fragment Programs	Description
`GL_MAX_PROGRAM_PARAMETERS_ARB`, `GL_MAX_PROGRAM_NATIVE_PARAMETERS_ARB`	96	24	Maximum number of parameters declared
`GL_MAX_PROGRAM_TEMPORARIES_ARB`, `GL_MAX_PROGRAM_NATIVE_TEMPORARIES_ARB`	12	16	Maximum number of temporaries declared
`GL_MAX_PROGRAM_ADDRESS_REGISTERS_ARB`, `GL_MAX_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB`	1		Maximum number of address registers declared (vertex programs only)
`GL_MAX_PROGRAM_ALU_INSTRUCTIONS_ARB`, `GL_MAX_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB`		48	Maximum number of ALU instructions declared (fragment programs only)
`GL_MAX_PROGRAM_TEX_INSTRUCTIONS_ARB`, `GL_MAX_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB`		24	Maximum number of texture instructions declared (fragment programs only)
`GL_MAX_PROGRAM_TEX_INDIRECTIONS_ARB`, `GL_MAX_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB`		4	Maximum number of texture indirections declared (fragment programs only)

To determine the resources actually consumed by the currently bound program of the specified target type, use one of the values in Table 13-19 for *pname*. There are two types of usage:

Overall usage — Overall usage is that of the program as written. The parameter names for overall usage are of the form `GL_PROGRAM*`.

Native usage — Native usage is for the program as compiled for the target hardware. In some cases, emulation of operations not directly supported by the hardware will consume additional resources. The parameter names for native usage are of the form `GL_PROGRAM_NATIVE*`.

Table 13-19 Program Resource Usage

Resource Usage Name (overall, native)	Description
`GL_PROGRAM_INSTRUCTIONS_ARB`, `GL_PROGRAM_NATIVE_INSTRUCTIONS_ARB`	Number of instructions used
`GL_PROGRAM_ATTRIBS_ARB`, `GL_PROGRAM_NATIVE_ATTRIBS_ARB`	Number of attributes used
`GL_PROGRAM_PARAMETERS_ARB`, `GL_PROGRAM_NATIVE_PARAMETERS_ARB`	Number of parameters used
`GL_PROGRAM_TEMPORARIES_ARB`, `GL_PROGRAM_NATIVE_TEMPORARIES_ARB`	Number of temporaries used
`GL_PROGRAM_ADDRESS_REGISTERS_ARB`, `GL_PROGRAM_NATIVE_ADDRESS_REGISTERS_ARB`	Number of address registers used (vertex programs only)
`GL_PROGRAM_ALU_INSTRUCTIONS_ARB`, `GL_PROGRAM_NATIVE_ALU_INSTRUCTIONS_ARB`	Number of ALU instructions used (fragment programs only)
`GL_PROGRAM_TEX_INSTRUCTIONS_ARB`, `GL_PROGRAM_NATIVE_TEX_INSTRUCTIONS_ARB`	Number of texture instructions used (fragment programs only)
`GL_PROGRAM_TEX_INDIRECTIONS_ARB`, `GL_PROGRAM_NATIVE_TEX_INDIRECTIONS_ARB`	Number of texture indirections used (fragment programs only)

To assist in determining if a program is running on the actual graphics hardware, call **glGetProgramivARB()** with *pname* set to `GL_PROGRAM_UNDER_NATIVE_LIMITS_ARB`. This returns `0` in *params* if the native resource consumption of the program currently

bound to *target* exceeds the number of available resources for any resource type and `1`, otherwise.

To determine the maximum number of program local parameters and program environment parameters that may be specified for *target*, use a *pname* of `GL_MAX_PROGRAM_LOCAL_PARAMETERS_ARB` or `GL_MAX_PROGRAM_ENV_PARAMETERS_ARB`, respectively.

To determine the maximum number of generic vertex attributes that may be specified for vertex programs, call **glGetIntegerv()** with a *pname* of `GL_MAX_VERTEX_ATTRIBS_ARB`.

To determine the maximum number of generic matrices that may be specified for programs, call **glGetIntegerv()** with a *pname* of `GL_MAX_PROGRAM_MATRICES_ARB`. At least 8 program matrices are guaranteed to be supported. To determine the maximum stack depth for generic program matrices, call **glGetIntegerv()** with a *pname* of `GL_MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB`. The maximum generic matrix stack depth is guaranteed to be at least `1`.

To determine properties of generic matrices, rather than extending **glGet*()** to accept the `GL_MATRIX0`+*n* terminology, additional parameter names are defined which return properties of the **current matrix** (as set with the **glMatrixMode()** function). The depth of the current matrix stack can be queried by calling **glGetIntegerv()** with a *pname* of `GL_CURRENT_MATRIX_STACK_DEPTH_ARB`, while the current matrix values can be queried by calling **glGetFloatv()** with a *pname* of `GL_CURRENT_MATRIX_ARB` or `GL_TRANSPOSE_CURRENT_MATRIX_ARB`. The functions return the 16 entries of the current matrix in column-major or row-major order, respectively.

Other Program Queries

In addition to program resource limits and usage, you can query for following information about the currently bound program:

- Program string length, program string format, and program string name
- Source text
- Parameters of the generic vertex attribute array pointers

Program String Length, Program String Format, and Program String Name

Calling **glGetProgramivARB()** with a *pname* of `GL_PROGRAM_LENGTH_ARB`, `GL_PROGRAM_FORMAT_ARB`, or `GL_PROGRAM_BINDING_ARB`, returns one integer reflecting the program string length (in GLubytes), program string format, and program name, respectively, for the program object currently bound to target.

Source Text

Making the following call returns the source text for the program bound to target in the array string:

```
void glGetProgramStringARB(GLenum target, GLenum pname, GLvoid *string);
```

The argument *pname* must be `GL_PROGRAM_STRING_ARB`. The size of string must be at least the value of `GL_PROGRAM_LENGTH_ARB` queried with **glGetProgramivARB()**. The program string is always returned using the format given when the program string was specified.

Parameters of the Generic Vertex Attribute Array Pointers

You can query the parameters of the generic vertex attribute array pointers by calling one of the following commands:

```
void glGetVertexAttribdvARB(GLuint index, GLenum pname,
                            GLdouble *params);
void glGetVertexAttribfvARB(GLuint index, GLenum pname, GLfloat *params);
void glGetVertexAttribivARB(GLuint index, GLenum pname, GLint *params);
```

The *pname* value must be one of the following:

- `GL_VERTEX_ATTRIB_ARRAY_ENABLED_ARB`
- `GL_VERTEX_ATTRIB_ARRAY_NORMALIZED_ARB`
- `GL_VERTEX_ATTRIB_ARRAY_SIZE_ARB`
- `GL_VERTEX_ATTRIB_ARRAY_STRIDE_ARB`
- `GL_VERTEX_ATTRIB_ARRAY_TYPE_ARB`

Bound generic vertex array pointers can be queried by making the following call:

```
void glGetVertexAttribPointervARB(GLuint index, GLenum pname,
                                  GLvoid **pointer);
```

Sample Code

These examples are intended primarily to show complete vertex and fragment programs using a range of instructions and input. The OpenGL programming required to set up and execute these programs on sample geometry are not included.

Sample Vertex Program

The following vertex program implements a simple ambient, specular, and diffuse infinite lighting computation with a single light and an eye-space normal:

```
!!ARBvp1.0
ATTRIB iPos         = vertex.position;
ATTRIB iNormal      = vertex.normal;
PARAM  mvinv[4]     = { state.matrix.modelview.invtrans };
PARAM  mvp[4]       = { state.matrix.mvp };
PARAM  lightDir     = state.light[0].position;
PARAM  halfDir      = state.light[0].half;
PARAM  specExp      = state.material.shininess;
PARAM  ambientCol   = state.lightprod[0].ambient;
PARAM  diffuseCol   = state.lightprod[0].diffuse;
PARAM  specularCol  = state.lightprod[0].specular;
TEMP   xfNormal, temp, dots;
OUTPUT oPos         = result.position;
OUTPUT oColor       = result.color;

# Transform the vertex to clip coordinates.
DP4   oPos.x, mvp[0], iPos;
DP4   oPos.y, mvp[1], iPos;
DP4   oPos.z, mvp[2], iPos;
DP4   oPos.w, mvp[3], iPos;

# Transform the normal to eye coordinates.
DP3   xfNormal.x, mvinv[0], iNormal;
DP3   xfNormal.y, mvinv[1], iNormal;
DP3   xfNormal.z, mvinv[2], iNormal;

# Compute diffuse and specular dot products and use LIT to compute
# lighting coefficients.
DP3   dots.x, xfNormal, lightDir;
DP3   dots.y, xfNormal, halfDir;
MOV   dots.w, specExp.x;
LIT   dots, dots;
```

```
# Accumulate color contributions.
MAD   temp, dots.y, diffuseCol, ambientCol;
MAD   oColor.xyz, dots.z, specularCol, temp;
MOV   oColor.w, diffuseCol.w;
END
```

Sample Fragment Programs

The following fragment program shows how to perform a simple modulation between the interpolated fragment color from rasterization and a single texture:

```
!!ARBfp1.0
ATTRIB tex = fragment.texcoord;      # First set of texture coordinates
ATTRIB col = fragment.color.primary; # Diffuse interpolated color

OUTPUT outColor = result.color;

TEMP tmp;

TXP tmp, tex, texture, 2D;           # Sample the texture

MUL outColor, tmp, col;              # Perform the modulation

END
```

The following fragment program simulates a chrome surface:

```
!!ARBfp1.0

########################
# Input Textures:
#----------------------
# Texture 0 contains the default 2D texture used for general mapping
# Texture 2 contains a 1D pointlight falloff map
# Texture 3 contains a 2D map for calculating specular lighting
# Texture 4 contains normalizer cube map
#
# Input Texture Coordinates:
#----------------------
# TexCoord1 contains the calculated normal
# TexCoord2 contains the light to vertex vector
# TexCoord3 contains the half-vector in tangent space
# TexCoord4 contains the light vector in tangent space
```

```
# TexCoord5 contains the eye vector in tangent space
########################

TEMP     NdotH, lV, L;

ALIAS    diffuse  = L;
PARAM       half  = { 0.5, 0.5, 0.5, 0.5 };
ATTRIB   norm_tc  = fragment.texcoord[1];
ATTRIB     lv_tc  = fragment.texcoord[2];
ATTRIB   half_tc  = fragment.texcoord[3];
ATTRIB  light_tc  = fragment.texcoord[4];
ATTRIB    eye_tc  = fragment.texcoord[5];
OUTPUT      oCol  = result.color;

TEX     L, light_tc, texture[4], CUBE; # Sample cube map normalizer

# Calculate diffuse lighting (N.L)
SUB     L, L, half;              # Bias L and then multiply by 2
ADD     L, L, L;
DP3     diffuse, norm_tc, L;     # N.L

# Calculate specular lighting component { (N.H), |H|^2 }
DP3     NdotH.x, norm_tc, half_tc;
DP3     NdotH.y, half_tc, half_tc;

DP3     lV.x, lv_tc, lv_tc;     # lV = (|light to vertex|)^2

#############
# Pass 2
#############

TEMP     base, specular;
ALIAS    atten  = lV;

TEX      base,     eye_tc, texture[0], 2D; # sample enviroment map
using eye vector
TEX      atten,    lV,     texture[2], 1D; # Sample attenuation map
TEX      specular, NdotH,  texture[3], 2D; # Sample specular NHHH map=
(N.H)^256

# specular = (N.H)^256 * (N.L)
# this ensures a pixel is only lit if facing the light (since the
specular
# exponent makes negative N.H positive we must do this)
MUL      specular, specular, diffuse;
```

```
# specular = specular * environment map
MUL      specular, base, specular;

# diffuse = diffuse * environment map
MUL      diffuse, base, diffuse;

# outColor = (specular * environment map) + (diffuse * environment map)
ADD      base, specular, diffuse;

# Apply point light attenutaion
MUL      oCol, base, atten.r;

END
```

Errors

If a program fails to load because it contains an error when **glBindProgramARB()** is called or because it would exceed the resource limits of the implementation, a `GL_INVALID_OPERATION` error is generated. Calling **glGetIntegerv()** with a *pname* of `GL_PROGRAM_ERROR_POSITION_ARB` will return the byte offset into the currently bound program string at which the error was detected, and calling **glGetString()** with *pname* `GL_PROGRAM_ERROR_STRING_ARB` will return a string describing the error (for example, a compiler error message).

If the currently bound vertex or fragment program does not contain a valid program and the corresponding vertex or fragment program mode is enabled, a GL_INVALID_OPERATION error is generated whenever **glBegin()**, **glRasterPos*()**, or any drawing command that performs an explicit **glBegin()**, such as **glDrawArrays()**, is called.

Under the following conditions, `GL_INVALID_VALUE` errors will be generated if the specified index exceeds the implementation limit for the number of attributes, program local parameters, program environment parameters, etc.:

- When specifying vertex attribute indices in immediate-mode or vertex array calls
- When specifying program parameter indices in specification or query calls and in similar calls

New Functions

The ARB_vertex_program and ARB_fragment_program extensions introduce the following functions:

```
glBindProgramARB()
glDeleteProgramsARB()
glDisableVertexAttribArrayARB()
glEnableVertexAttribArrayARB()
glGenProgramsARB()
glGetProgramEnvParameterdvARB()
glGetProgramEnvParameterfvARB()
glGetProgramLocalParameterdvARB()
glGetProgramLocalParameterfvARB()
glGetProgramStringARB()
glGetProgramivARB()
glGetVertexAttribPointervARB()
glGetVertexAttribdvARB()
glGetVertexAttribfvARB()
glGetVertexAttribivARB()
glIsProgramARB()
glProgramEnvParameter4dARB()
glProgramEnvParameter4dvARB()
glProgramEnvParameter4fARB()
glProgramEnvParameter4fvARB()
glProgramLocalParameter4dARB()
glProgramLocalParameter4dvARB()
glProgramLocalParameter4fARB()
glProgramLocalParameter4fvARB()
glProgramStringARB()
glVertexAttrib1dARB()
glVertexAttrib1dvARB()
glVertexAttrib1fARB()
glVertexAttrib1fvARB()
glVertexAttrib1sARB()
glVertexAttrib1svARB()
glVertexAttrib2dARB()
glVertexAttrib2dvARB()
glVertexAttrib2fARB()
glVertexAttrib2fvARB()
glVertexAttrib2sARB()
glVertexAttrib2svARB()
glVertexAttrib3dARB()
glVertexAttrib3dvARB()
glVertexAttrib3fARB()
```

```
glVertexAttrib3fvARB()
glVertexAttrib3sARB()
glVertexAttrib3svARB()
glVertexAttrib4NbvARB()
glVertexAttrib4NivARB()
glVertexAttrib4NsvARB()
glVertexAttrib4NubARB()
glVertexAttrib4NubvARB()
glVertexAttrib4NuivARB()
glVertexAttrib4NusvARB()
glVertexAttrib4bvARB()
glVertexAttrib4dARB()
glVertexAttrib4dvARB()
glVertexAttrib4fARB()
glVertexAttrib4fvARB()
glVertexAttrib4ivARB()
glVertexAttrib4sARB()
glVertexAttrib4svARB()
glVertexAttrib4ubvARB()
glVertexAttrib4uivARB()
glVertexAttrib4usvARB()
glVertexAttribPointerARB()
```

The Legacy Vertex and Fragment Program Extensions

In addition to the ARB_vertex_program and ARB_fragment_program extension, Onyx4 and Silicon Graphics Prism systems also support the following set of ATI vendor extensions for vertex and fragment programming:

- ATI_fragment_shader
- EXT_vertex_shader

These two extensions, developed prior to the ARB extensions, are included only for support of legacy applications being ported from other platforms. They supply no functionality not present in ARB_vertex_program and ARB_fragment_program and are not as widely implemented. Whenever writing new code using vertex or fragment programs, always use the ARB extensions.

How to Use the Legacy Extensions

Since these are legacy extensions, they are not documented in detail here. This section only describes how the legacy extensions map onto the corresponding ARB extensions.

EXT_vertex_shader	Allows an application to define vertex programs that are functionally comparable to ARB_vertex_program programs.
ATI_fragment_shader	Allows an application to define fragment programs that are functionally comparable to ARB_fragment_program programs.

Instead of specifying a program string, each legacy program instruction is a function call specifying instruction parameters.

New Functions

The ATI_fragment_shader and EXT_vertex_shader extensions introduce the following set of functions:

```
glBindLightParameterEXT()
glBindMaterialParameterEXT()
glBindParameterEXT()
glBindTexGenParameterEXT()
glBindTextureUnitParameterEXT()
glGenFragmentShadersATI()
glGenSymbolsEXT()
glGenVertexShadersEXT()
glAlphaFragmentOp1ATI()
glAlphaFragmentOp2ATI()
glAlphaFragmentOp3ATI()
glBeginFragmentShaderATI()
glBeginVertexShaderEXT()
glBindFragmentShaderATI()
glBindVertexShaderEXT()
glColorFragmentOp1ATI()
glColorFragmentOp2ATI()
glColorFragmentOp3ATI()
glDeleteFragmentShaderATI()
glDeleteVertexShaderEXT()
glDisableVariantClientStateEXT()
glEnableVariantClientStateEXT()
glEndFragmentShaderATI()
glEndVertexShaderEXT()
```

```
glExtractComponentEXT()
glGetInvariantBooleanvEXT()
glGetInvariantFloatvEXT()
glGetInvariantIntegervEXT()
glGetLocalConstantBooleanvEXT()
glGetLocalConstantFloatvEXT()
glGetLocalConstantIntegervEXT()
glGetVariantBooleanvEXT()
glGetVariantFloatvEXT()
glGetVariantIntegervEXT()
glGetVariantPointervEXT()
glInsertComponentEXT()
glIsVariantEnabledEXT()
glPassTexCoordATI()
glSampleMapATI()
glSetFragmentShaderConstantATI()
glSetInvariantEXT()
glSetLocalConstantEXT()
glShaderOp1EXT()
glShaderOp2EXT()
glShaderOp3EXT()
glSwizzleEXT()
glVariantPointerEXT()
glVariant{bsifdubusui}vEXT()
glWriteMaskEXT()
```

OpenGL Tools

This chapter explains how to work with the following OpenGL tools:

- "ogldebug—The OpenGL Debugger" on page 386 lets you use a graphical user interface to trace and examine OpenGL calls.
- "The OpenGL Character Renderer (GLC)" on page 400 lets you render characters in OpenGL programs.
- "The OpenGL Stream Utility (GLS)" on page 400 is a facility for encoding and decoding streams of 8-bit bytes that represent sequences of OpenGL commands.
- "glxinfo—The glx Information Utility" on page 402 provides information on GLX extensions and OpenGL capable visuals, and the OpenGL renderer of an X server.

The first section describes platform limitations.

Platform Notes

Currently, the `ogldebug`, GLC, and GLS tools are only supported on SGI IRIX systems while the `glxinfo` tool is supported on both IRIX and Linux systems.

Depending on customer demand, the tools `ogldebug` and GLS may be supported on Linux systems in the future. The GLC tool is obsolete, although still supported on IRIX systems. There are several alternative, open source toolkits for high-quality font rendering in OpenGL. You should migrate applications to one of those alternatives.

ogldebug—The OpenGL Debugger

This section explains how to debug graphics applications with the OpenGL debugging tool `ogldebug`. The following topics are described:

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, which contains comments and performance hints. You can convert it to valid 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 in the folllowing manner:

- You invoke ogldebug for an application using the appropriate command line options (see "ogldebug Command-Line Options" on page 388).
- 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 describes 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** option from the **Toolchest** menu on the IRIX desktop or execute *swmgr* from the command line. Consider also installing *gl_dev.man.ogldebug* to have access to the man page.

`ogldebug` Command-Line Options

The `ogldebug` version that is shipped with IRIX 6.5 has a number of command-line options, which are shown in Table 14-1. The options are also listed in the `ogldebug` man page.

Table 14-1 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** *options o32program_name program_options*
n32	% **ogldebug32** *options n32program_name program_options*
64	% **ogldebug64** *options 64program_name program_options*

The following describes the parameters:

options	Any of the options listed under "ogldebug Command-Line Options."
program_name	The name of your (executable) application.
program_options	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`.

The `ogldebug` tool 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 14-1) lets you interact with your application's current process and displays information about the process.

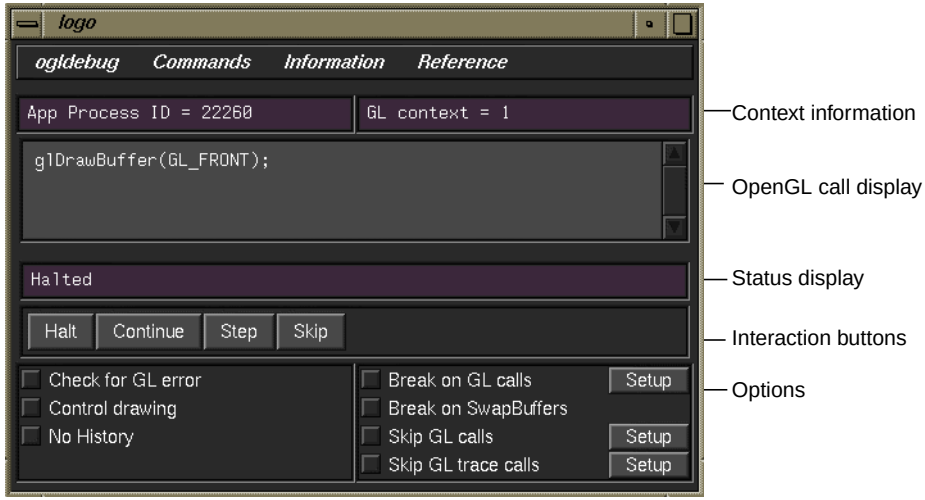


Figure 14-1 `ogldebug` Main Window

The following are display areas below the menu bar:

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, described in “Using Check boxes” on page 392.

Interacting With ogldebug

This section provides more detailed information on working with `ogldebug`. The following topics are described:

- "Commands for Basic Interaction"
- "Using Check boxes"

Additional information is available in the sections "Creating a Trace File to Discover OpenGL Problems" on page 393 and "Using Menus to Interact With ogldebug" on page 395.

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 14-2 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 Check boxes

The check boxes at the bottom of the `ogldebug` window allow finer control over how information is collected. Check boxes let you determine when a break occurs and which API calls you want to skip.

Table 14-3 explains what happens for each of these boxes if it is checked.

Table 14-3 `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 you to inspect drawing in progress (forces front buffer rendering). Also, allows you 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 14-2). In the **Break on GL calls Setup** box, **glFlush()** is selected by default but is not active unless the **Break on GL calls** check box is selected.
Break on SwapBuffers	Halts on calls that swap buffers. There is no window-system-independent call that swaps buffers; the debugger halts on the appropriate call for each platform (for example. **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 do not want traced.

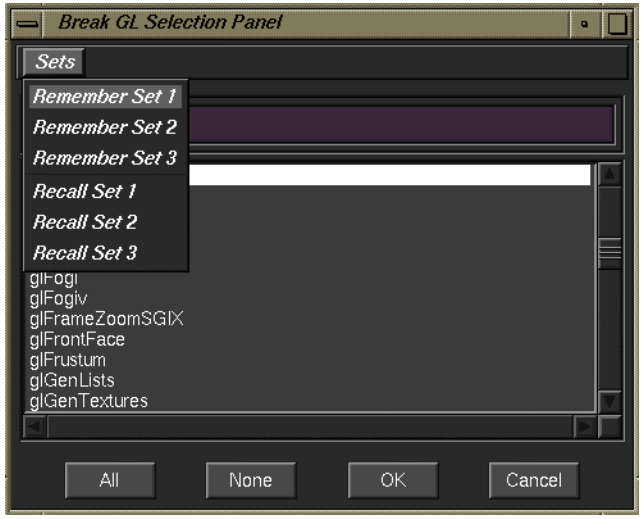


Figure 14-2 Setup Panel

Figure 14-2 shows a setup panel. Inside any setup panel, you can use the standard `Shift`, `Ctrl`, and `Shift+Ctrl` 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. The following is an example of how to collect one frame of OpenGL calls:

1. Launch `ogldebug` as follows:

   `% ogldebug` *your_program_name*

   Be sure to use the appropriate options,; see “ogldebug Command-Line Options” on page 388.

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**.

   The `ogldebug` tool 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 to show 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. However, comments 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,
             2.589596E-29});
glsString("Info", "For best performance, set up material parameters
           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 OpenGL 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 the no-history feature to speed things up.

Using a Configuration File

As you work with ogldebug, you may find it useful to save and reload certain settings. 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 resulting dialog box.
- To load settings, choose **Load Configuration** from the **File** menu, then select a file using the resulting dialog box.

Using Menus to Interact With ogldebug

This section describes how you can interact with ogldebug using menus. The following tasks are described:

Using the File Menu to Interact With ogldebug

The **File** menu (shown in Figure 14-3) gives version information, lets you save and reload a configuration file, and stops ogldebug.

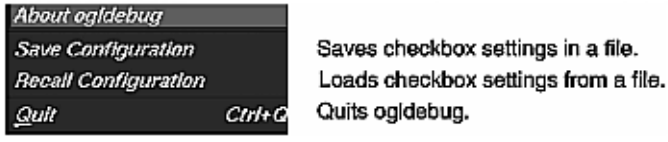


Figure 14-3 ogldebug **File** Menu

Using the Commands Menu to Interact With Your Program

The **Commands** menu gives you access to commands that control `ogldebug` execution.The commands are described in "Interacting With ogldebug" on page 391.

Figure 14-4 `ogldebug` **Commands** 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.

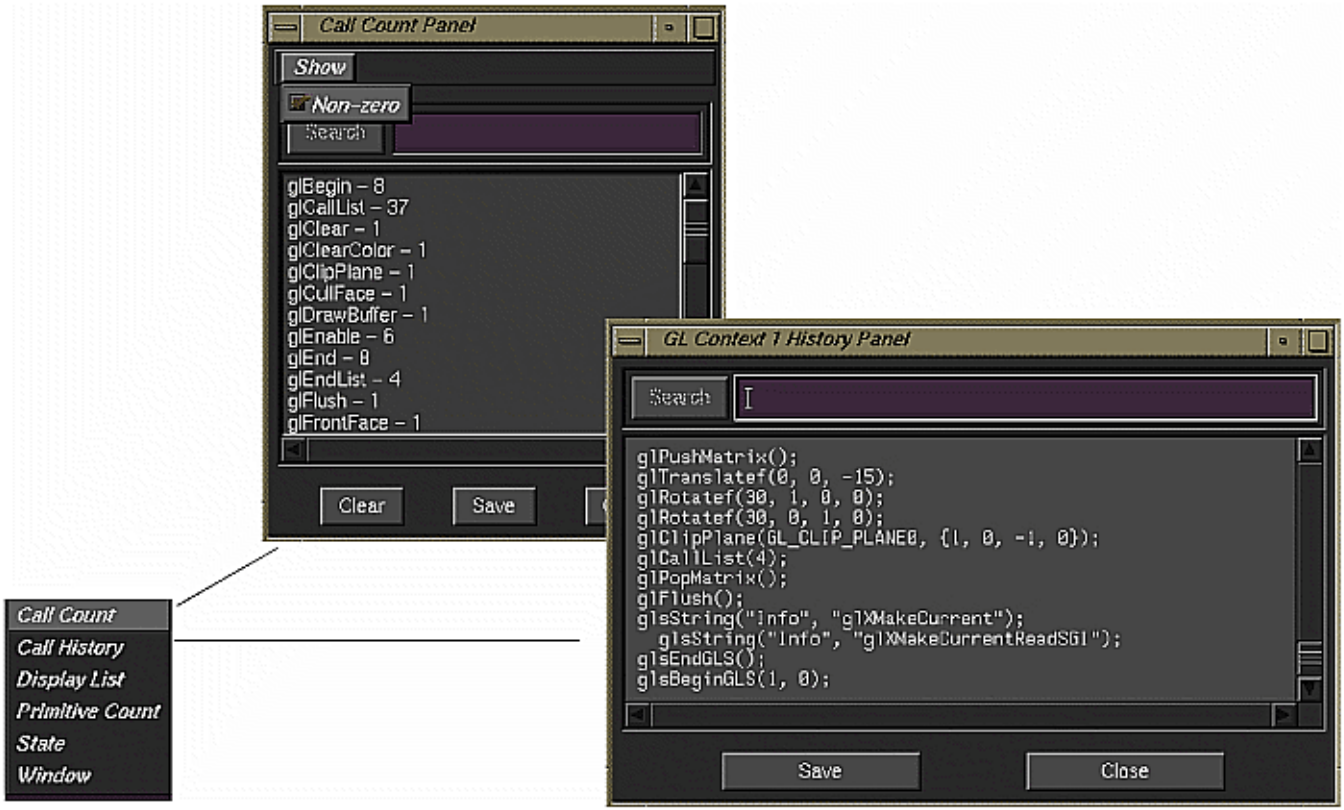


Figure 14-5 **Information** Menu Commands (First Screen)

The following is 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 show 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).

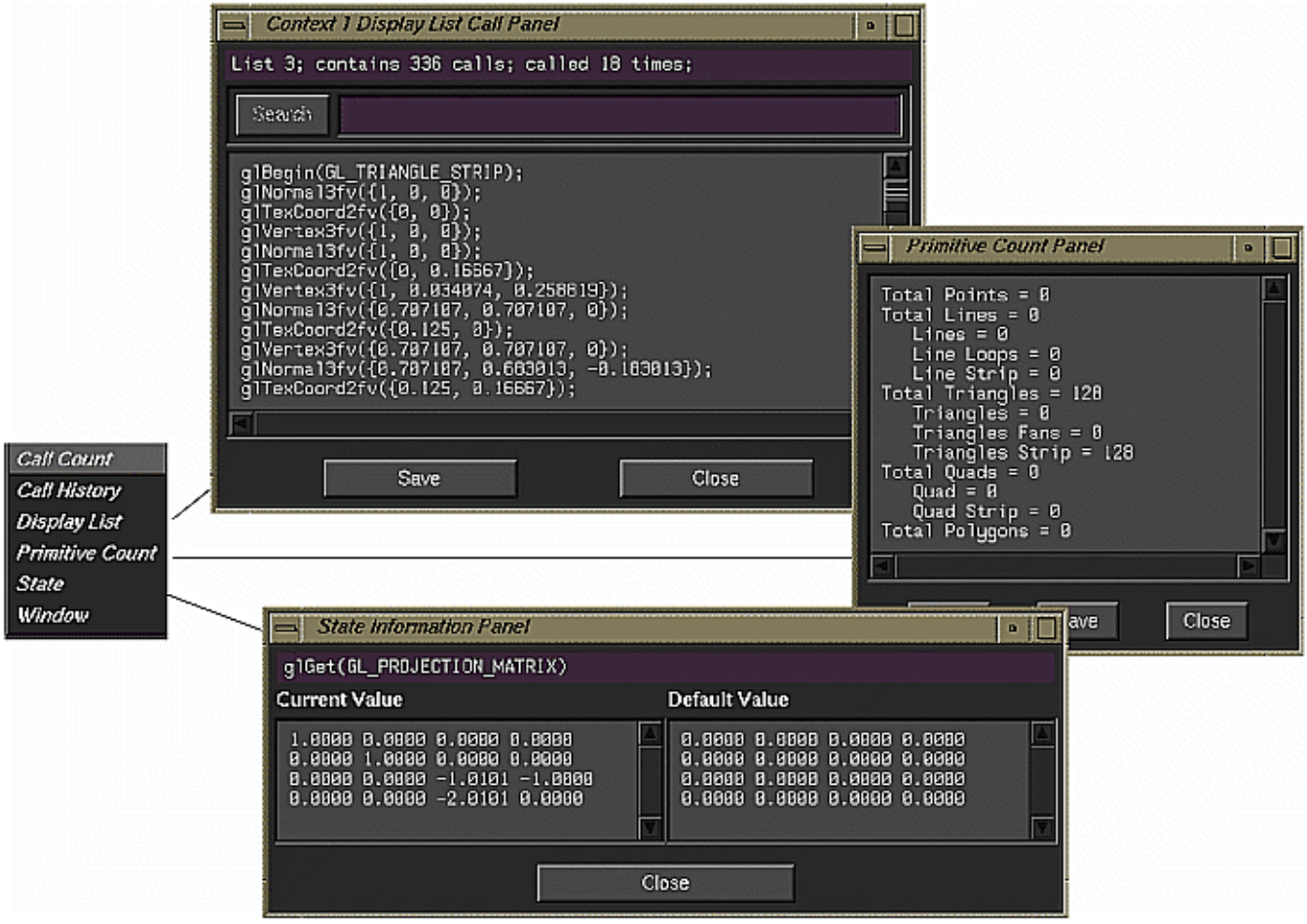


Figure 14-6 **Information** Menu Commands (Second Screen)

The following 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-empty display lists.

Primitive Count Provides the number of all primitives sent by the application so far (for example, quads, 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 show 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 14-7).

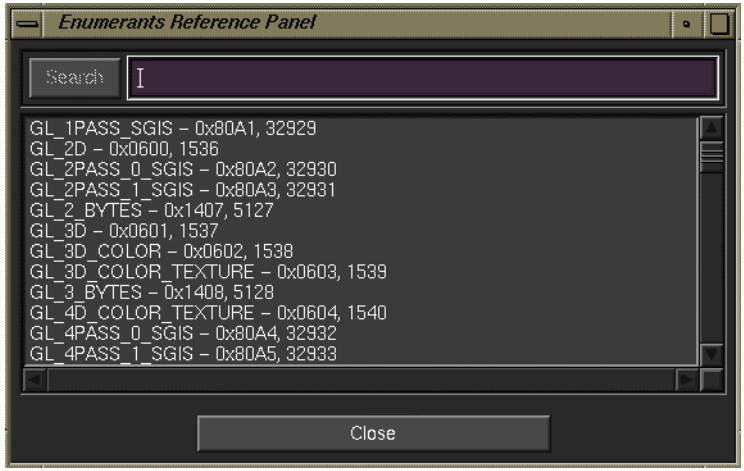


Figure 14-7 **Enumerants** Window

The OpenGL Character Renderer (GLC)

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 international characters.

For a basic discussion of GLC and a list of notes and known bugs for the current implementation, see the `glcintro` man 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 the following file:

```
/usr/share/src/OpenGL/teach/glc/glcspec.ps
```

The OpenGL Stream Utility (GLS)

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 describes “glscat Utility” on page 401, which allows you to concatenate GLS streams.

OpenGL Stream Utility Overview

GLS can be used for a variety of purposes—the following, 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 through 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. Therefore, it is 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 independent of platform and window system. 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 more information, see the `glsintro` man 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.

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

The `glxinfo` utility 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 the command or see the `glxinfo` man page.

Tuning Graphics Applications: Fundamentals

Tuning your software can make it use hardware capabilities more effectively. Even the fastest machine can render only as fast as the application can drive it. Simple changes in application code can often make a dramatic difference in rendering time. In addition, Silicon Graphics systems let you make trade-offs between image quality and performance for your application.

This chapter looks at tuning graphics applications. Using the following sections, this chapter describes pipeline tuning as a conceptual framework for tuning graphics applications and introduces some other fundamentals of tuning:

- "General Tips for Debugging Graphics Programs" on page 404
- "Specific Problems and Troubleshooting" on page 405
- "About Pipeline Tuning" on page 409
- "Tuning Animation" on page 418
- "Taking Timing Measurements" on page 413

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, see the following chapters in this book:

- Chapter 16, "Tuning the Pipeline"
- Chapter 17, "Tuning Graphics Applications: Examples"
- Chapter 18, "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 OpenGL Performer rendering toolkit. See "Maximizing Performance With OpenGL Performer" on page 7.

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; you must determine whether an error occurred. 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" on page 386 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.

The following are some areas that frequently experience 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 invalid. The exact list of allowable commands is given in the man 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 describes some specific problems frequently encountered by OpenGL users. Note that one generally useful approach is to experiment with an `ogldebug` trace of the first few frames. See "Creating a Trace File to Discover OpenGL Problems" on page 393. This section covers the following problems:

Blank Window

A common problem encountered in graphics programming is a blank window. If you find your display does not 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 do not 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, ensure that the depth buffer is cleared as appropriate. See also "Depth Buffering Problems" on page 406.
- Check the aspect ratio of the viewing frustrum. Do not 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

The following rotation and translation areas might be trouble spots:

- Z axis direction

  Remember that by default you start by 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 and then rotating an object yields a different result than first rotating and 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 do the following:

- 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`.

- To guarantee that your program is portable, always ask for a depth buffer explicitly when requesting a visual or framebuffer configuration.

Animation Problems

The following two areas might be animation problem areas:

- 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 do not 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

If you are having lighting problems, try one or more of the followig actions:

- 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

The following items identify possible problem sources with the X Window system:

- OpenGL and the X Window System have different notions of the y direction. OpenGL has the origin (0, 0) in the lower left corner of the window; X has the origin 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

If you are having problems writing pixels or textures, ensure that 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 value for `GL_UNPACK_ALIGNMENT` is `4`. It should be `1` in the preceding case. If this value is not set correctly, the image looks sheared.

The same thing applies to textures.

System-Specific Problems

Ensure you do not exceed implementation-specific resource limits such as maximum projection stack depth. In general, consult the documentation for your platform for likely problem areas.

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; therefore, it is important to understand which elements in the graphics pipeline are a program's bottlenecks.

A Three-Stage Model of the Graphics Pipeline

The graphics pipeline in all Silicon Graphics systems consists of three conceptual stages (see Figure 15-1). 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.

The following are the three stages of the model:

The CPU subsystem	The application program running on the CPU, feeding commands to the graphics subsystem.

The geometry subsystem	The per-polygon operations, such as coordinate transformations, lighting, texture coordinate generation, and clipping (may be hardware accelerated).
The raster system	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.

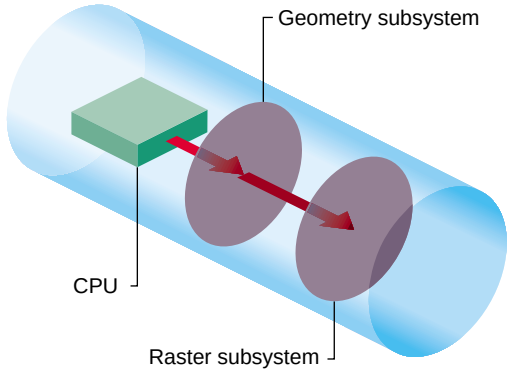


Figure 15-1 A Three-Stage Model of the Graphics Pipeline

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 does not 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" on page 421.

- Geometry bottlenecks

  Programs that create bottlenecks in the geometry (per-polygon) stage are called *transform-limited*. To test for bottlenecks in geometry operations, change the program so that the application code runs at the same speed and the same number of pixels are filled, but the geometry work is reduced. For example, if you are using lighting, call **glDisable()** with a `GL_LIGHTING` argument to turn off lighting temporarily. If performance improves, your application has a per-polygon bottleneck. For more information, see "Tuning the Geometry Subsystem" on page 440.

- 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 does not 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" on page 448.

Usually, the following order of operations is the most expedient:

1. First determine if your application is CPU-limited using `gr_osview` or `top` and checking whether the CPU usage is near 100%. The `gr_osview` program (supported only on SGI IRIX systems) 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-rate-limited by shrinking the window.
3. If the application is neither CPU-limited nor fill-rate-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 17, "Tuning Graphics Applications: Examples."

Factors Influencing Performance

Pipeline tuning is described in detail in Chapter 16, "Tuning the Pipeline." Table 15-1 provides an overview of factors that may limit rendering performance and the stages of the pipeline involved.

Table 15-1 Factors Influencing Performance

Performance Parameter	Pipeline Stage
Amount of data per polygon	All stages
Time of application overhead	CPU subsystem (application)
Transform rate and 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 applications such as `top`, *`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.

  The function **gettimeofday()** provides a convenient interface to system clocks with enough resolution to measure graphics performance over several frames. On IRIX systems, 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 do not 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” on page 386 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" on page 411.
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.

Try some of the suggestions presented in the following chapter on a small program that you understand and use benchmarks to observe the effects. Figure 15-2 shows how you may actually go through the process of benchmarking and reducing bottlenecks several times. This is also demonstrated by the example presented in Chapter 17, "Tuning Graphics Applications: Examples."

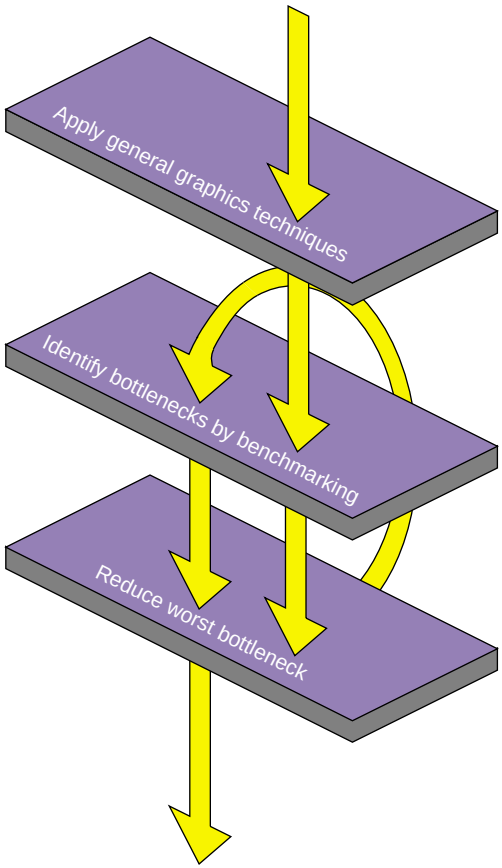


Figure 15-2 Flowchart of the Tuning Process

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 and 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 15-1 on page 413.

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 for *n+1* vertical retraces 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 and then benchmark how many frames you can draw. If you do not, 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” on page 413.

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.

Tuning the Pipeline

Providing code fragments and examples as appropriate, this chapter presents a variety of techniques for optimizing the different parts of the pipeline. The following topics are used:

CPU Tuning: Basics

The first stage of the rendering pipeline is the traversal of the data and sending the current rendering data to the rest of the pipeline. In theory, the entire rendering database (scene graph) must be traversed in some fashion for each frame because both scene content and viewer position can be dynamic.

To get the best possible CPU performance, use the following 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.

- On IRIX systems, always compile for the n32 ABI, instead of the obsolete o32 ABI. n32 is now the default for the IRIX compilers.
- 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 does not become the bottleneck.

Immediate Mode Drawing Versus Display Lists and Vertex Buffer Objects

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. The following are three cases to consider:

- If you create models that never change and are used only for drawing, then OpenGL display lists or vertex buffer objects are the right representation.

  Display lists can be optimized in hardware-specific ways, loaded into dedicated display list storage in the graphics subsystem, downloaded to on-board dlist RAM, and so on. See “CPU Tuning: Display Lists” on page 424 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 vertext buffer objects or vertex arrays are the right representation.

  Vertex arrays are relatively compact and have modest impact on the cache. Software renderers can process the vertices in batches; hardware renderers can process a few triangles at a time to maximize parallelism. As long as the vertex arrays can be retained from frame to frame so that 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” on page 442 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 do the following:

  – To maximize parallelism for hardware renderers

  – To optimize your database for the other computations you need to perform

  – To reduce 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 database— for example, for morphing operations where the number of vertices is subject to change or for progressive refinement. See "CPU Tuning: Immediate Mode Drawing" on page 425 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 do not 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 10) 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.

Display lists do have the following 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 need for the data

other than 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, and 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.
- The compile time for display lists may be significant.

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 traversed again.

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 the 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 for the 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 18, "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 the following 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 the following performance concerns and should be used with care:

- 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, to cause 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; therefore, traversing hierarchical structures can access a costly number of pages.

- Complex operations may need access to both the geometric data and other scene information, complicating the data structure.

- Caching behavior is often difficult to predict for dynamic hierarchical data structures.

The following are rules for optimizing data organization:

- 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 do not 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. Hot spots are instructions in the display loop that are executed many times every frame. 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 (Meshes and 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, as shown in the following:

  Bad `glVertex3fv(object->data->vert);`

  OK `glVertex3fv(dataptr->vert);`

  Good `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

The following are areas for optimizing your 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 current MIPS processors, 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 will not 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 run-time typecasting of parameters:

  ```
  void drawit(float f, int count)
  {
  }
  ```

- **Multiple primitives.** Send multiple primitives between **glBegin()/glEnd()** pairs whenever possible:

```
glBegin(GL_TRIANGLES)
....
..../* many triangles */
....
glEnd

glBegin(GL_QUADS)
....
..../* many quads */
....
glEnd
```

Using Specialized Drawing Subroutines and Macros

This section describes several ways to improve performance by making appropriate choices about display modes, geometry, and so on.

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 the `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 (Meshes and Vertex Loops)

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 describes preprocessing meshes and 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, as shown in the following:

```
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 platform. For more information, see Chapter 18, "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.

The following are the areas of concern:

- 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:

- "Minimizing Lookups"
- "Minimizing Cache Misses"
- "Measuring Cache-Miss and Page-Fault Overhead"

Minimizing Lookups

To minimize page lookups, 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 caches 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. In 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 multiprocessing application. This can happen with multiple processes traversing very large data sets. See “Immediate Mode Drawing Versus Display Lists and Vertex Buffer Objects” on page 422 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. Perform 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` man 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 the following 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 do the following:

- To determine where the application is demonstrably graphics-limited
- To 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. The result is worse performance than the original organization.

Examining Assembly Code

When tuning inner rendering loops, examining assembly code can be helpful. You need not be an expert assembly coder to interpret the results. Just looking at the number of extra instructions required for an apparently innocuous operation is often informative.

On IRIX systems, use the *dis* command to disassemble optimized code for a given procedure and to 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 the *cc* command produces a `.s` file of assembly output, complete with your original comments.

On Silicon Graphics Prism systems, use the `objdump` –d [–S] command instead of the `dis` command. The –S option is available on the `gcc` command but comments are not included.

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.

  The sprite extension can be used for billboards on certain platforms; see “SGIX_sprite—The Sprite Extension” on page 250.

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 the following techniques for tuning the geometry subsystem:

- "Using Peak-Performance Primitives for Drawing"
- "Using Vertex Arrays"
- "Using Display Lists Appropriately"
- "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 the following 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()** block are required to achieve peak rates (but this can vary depending on your hardware platform). 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 18, "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 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 the overhead for subroutine calls.
- The **glDrawElements()** call reduces the overhead for subroutine calls and also reduces per-vertex calculations because vertices are reused.

Using Display Lists Appropriately

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” on page 424).

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 your system. 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 your system has 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. Use the following 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 and others offer sophisticated effects with some performance penalty. For some features, the penalties may vary depending on your hardware. 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 will not 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
  - Ensure that `GL_LIGHT_MODEL_LOCAL_VIEWER` is set to `GL_FALSE` in **glLightModel()** (the default).
  - Ensure that `GL_LIGHT_MODEL_TWO_SIDE` is 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 modelview 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:

- Do not 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 do not 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 the following call:

  ```
  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.

  Flat shading 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. Use the following 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 provide simplified geometry with 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, make the following 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 describes 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/frontface removal. For example, if you are drawing a sphere, half of its polygons are backfacing at any given time. Backface/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 describes the following ways to 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 the 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 your system.

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.

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; the sky, ground, and 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 does not 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 the following function:

```
glDepthFunc(GL_ALWAYS)
```

Balancing Polygon Size and Pixel Operations

The pipeline is generally optimized for polygons that have 10 pixels on a side. However, you may need to work with polygons larger or smaller than that depending on the other operations taking place 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, do not 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

The following are other ways to improve fill-limited drawing:

- 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, use the following guidelines:

- 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, do not clear each buffer independently. Instead, make the following 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 17-1 on page 457 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 that 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 a color matrix and a color mask to store four luminance values in the RGBA framebuffer. Use a color matrix and a 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().**

- Locate the bottleneck.

  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 a `GL_RGB5_A1` internal format.

  When either many processing modes or 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 distributes the setup cost over many pixels to achieve 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 overhead 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.

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, shows changes to the program, and describes the speed improvements that result. The two sections are titled as follows:

Drawing Pixels Fast

The code fragment in Example 17-1 illustrates how to set an OpenGL state so that subsequent calls to **glDrawPixels()** or **glCopyPixels()** will be fast.

Example 17-1 Drawing Pixels Fast

```
/*
 * Disable stuff that is 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.That code was omitted 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 16, “Tuning the Pipeline.” Consider a program that draws a lighted sphere, shown in Figure 17-1.

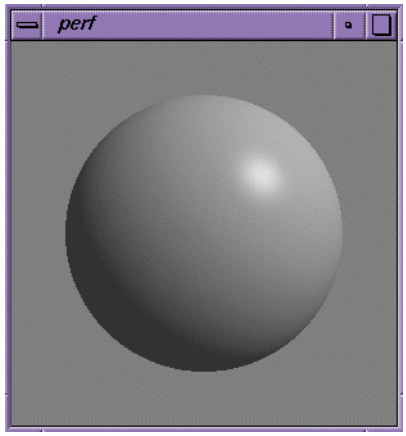


Figure 17-1 Lighted Sphere Created by perf.c

You can use the benchmarking framework in Appendix A, “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.

Example 17-2 Example Program—Performance Tuning

```
/*****************************************************************
   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 are not including time from some
 * previous uncompleted operation in the measurements. (It is not
 * foolproof, since you cannot eliminate context switches, but you can
 * assume the caller has taken care of that problem.)
*****************************************************************/
void
FinishDrawing(void) {
        glFinish();
        }


/*****************************************************************
 * WaitForTick - wait for beginning of next system clock tick; return
 * the time
*****************************************************************/
double
WaitForTick(void) {
        double start;
        double current;

        start = GetClock();

        /* Wait for next tick: */
        while ((current = GetClock()) == start)
                ;

        /* Start timing: */
        return current;
        }


/*****************************************************************
 * InitBenchmark - measure benchmarking overhead
 *
 * This should be done once before each risky change in the
 * benchmarking environment. A “risky” change is one that might
 * reasonably be expected to affect benchmarking overhead. (For
 * example, changing from a direct rendering context to an indirect
 * rendering context.)  If all measurements are being made on a single
 * rendering context, one call should suffice.
*****************************************************************/

void
```

```
InitBenchmark(void) {
        double runTime;
        long reps;
        double start;
        double finish;
        double current;

        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /* Wait for the pipe to clear: */
        FinishDrawing();

        /* Measure approximate overhead for finalization and timing
         * routines: */
        reps = 0;
        start = WaitForTick();
        finish = start + runTime;
        do {
                FinishDrawing();
                ++reps;
                } while ((current = GetClock()) < finish);

        /* Save the overhead for use by Benchmark(): */
        Overhead = (current - start) / (double) reps;
        }

/*********************************************************************
 * Benchmark--measure number of caller operations performed per second
 *
 * Assumes InitBenchmark() has been called previously, to initialize
 * the estimate for timing overhead.
*********************************************************************/
double
Benchmark(void (*operation)(void)) {
        double runTime;
        long reps;
        long newReps;
        long i;
        double start;
        double current;

        if (!operation)
                return 0.0;
        /* Select a run time appropriate for our timer resolution: */
```

```
        runTime = ChooseRunTime();

        /*
         * Measure successively larger batches of operations until you
         * find one that is long enough to meet our run-time target:
         */
        reps = 1;
        for (;;) {
                /* Run a batch: */
                FinishDrawing();
                start = WaitForTick();
                for (i = reps; i > 0; --i)
                        (*operation)();
                FinishDrawing();

                /* If we reached our target, get out of the loop: */
                current = GetClock();
                if (current >= start + runTime + Overhead)
                        break;

                /*
                 * Otherwise, increase the rep count and try to reach
                 * the target on the next attempt:
                 */
                if (current > start)
                        newReps = reps *(0.5 + runTime /
                                        (current - start - Overhead));
                else
                        newReps = reps * 2;
                if (newReps == reps)
                        reps += 1;
                else
                        reps = newReps;
                }

        /* Subtract overhead and return the final operation rate: */
        return (double) reps / (current - start - Overhead);
        }
/*************************************************************************
 * Test - the operation to be measured
 *
 * Will be run several times in order to generate a reasonably accurate
 * result.
*************************************************************************/
void
```

```
Test(void) {
        float latitude, longitude;
        float dToR = M_PI / 180.0;

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        for (latitude = -90; latitude < 90; ++latitude) {
                glBegin(GL_QUAD_STRIP);
                for (longitude = 0; longitude <= 360; ++longitude) {
                      GLfloat x, y, z;
                      x = sin(longitude * dToR) * cos(latitude * dToR);
                      y = sin(latitude * dToR);
                      z = cos(longitude * dToR) * cos(latitude * dToR);
                      glNormal3f(x, y, z);
                      glVertex3f(x, y, z);
                      x = sin(longitude * dToR) * cos((latitude+1) *
                                                              dToR);
                      y = sin((latitude+1) * dToR);
                        z = cos(longitude * dToR) * cos((latitude+1) *
                                                              dToR);
                      glNormal3f(x, y, z);
                      glVertex3f(x, y, z);
                      }
                glEnd();
                }
        }

/*****************************************************************
 * RunTest - initialize the rendering context and run the test
*****************************************************************/
void
RunTest(void) {
        static GLfloat diffuse[] = {0.5, 0.5, 0.5, 1.0};
        static GLfloat specular[] = {0.5, 0.5, 0.5, 1.0};
        static GLfloat direction[] = {1.0, 1.0, 1.0, 0.0};
        static GLfloat ambientMat[] = {0.1, 0.1, 0.1, 1.0};
        static GLfloat specularMat[] = {0.5, 0.5, 0.5, 1.0};

        if (Overhead == 0.0)
                InitBenchmark();

        glClearColor(0.5, 0.5, 0.5, 1.0);

        glClearDepth(1.0);
        glEnable(GL_DEPTH_TEST);
```

```
        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
        glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
        glLightfv(GL_LIGHT0, GL_POSITION, direction);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHTING);

        glMaterialfv(GL_FRONT, GL_AMBIENT, ambientMat);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specularMat);
        glMateriali(GL_FRONT, GL_SHININESS, 128);

        glEnable(GL_COLOR_MATERIAL);
        glShadeModel(GL_SMOOTH);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45.0, 1.0, 2.4, 4.6);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        gluLookAt(0,0,3.5,  0,0,0,  0,1,0);

        printf("%.2f frames per second\n", Benchmark(Test));
        }

/*****************************************************************
 * ProcessEvents - handle X11 events directed to our window
 *
 * Run the measurement each time we receive an expose event.
 * Exit when you receive a press of the Escape key.
 * Adjust the viewport and projection transformations when the window
 * changes size.
*****************************************************************/
void
ProcessEvents(Display* dpy) {
        XEvent event;
        Bool redraw = 0;

        do {
                char buf[31];
                KeySym keysym;

                XNextEvent(dpy, &event);
                switch(event.type) {
                        case Expose:
```

```
                                redraw = 1;
                                break;
                        case ConfigureNotify:
                                glViewport(0, 0,
                                        WindowWidth =
                                              event.xconfigure.width,
                                        WindowHeight =
                                              event.xconfigure.height);
                                redraw = 1;
                                break;
                        case KeyPress:
                                (void) XLookupString(&event.xkey, buf,
                                        sizeof(buf), &keysym, NULL);
                                switch (keysym) {
                                        case XK_Escape:
                                                exit(EXIT_SUCCESS);
                                        default:
                                                break;
                                        }
                                break;
                        default:
                                break;
                        }
                } while (XPending(dpy));

        if (redraw) RunTest();
        }

/*****************************************************************
 * Error - print an error message, then exit
*****************************************************************/
void
Error(const char* format, ...) {
        va_list args;

        fprintf(stderr, "%s:  ", ApplicationName);

        va_start(args, format);
        vfprintf(stderr, format, args);
        va_end(args);

        exit(EXIT_FAILURE);
        }

/*****************************************************************
```

```
 * main - create window and context, then pass control to ProcessEvents
******************************************************************/
int
main(int argc, char* argv[]) {
        Display *dpy;
        XVisualInfo *vi;
        XSetWindowAttributes swa;
        Window win;
        GLXContext cx;

        ApplicationName = argv[0];

        /* Get a connection: */
        dpy = XOpenDisplay(NULL);
        if (!dpy) Error("can't open display");

        /* Get an appropriate visual: */
        vi = glXChooseVisual(dpy, DefaultScreen(dpy),
                             VisualAttributes);
        if (!vi) Error("no suitable visual");

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

        /* Create a color map: */
        swa.colormap = XCreateColormap(dpy, RootWindow(dpy,
                                  vi->screen), vi->visual, AllocNone);

        /* Create a window: */
        swa.border_pixel = 0;
        swa.event_mask = ExposureMask | StructureNotifyMask |
                                                          KeyPressMask;
        win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0,
                       300, 300, 0, vi->depth, InputOutput, vi->visual,
                       CWBorderPixel|CWColormap|CWEventMask, &swa);
        XStoreName(dpy, win, "perf");
        XMapWindow(dpy, win);

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

        /* Handle events: */
        while (1) ProcessEvents(dpy);
        }
```

Testing for CPU Limitation

An application may be CPU-limited, geometry-limited, or fill-limited. Start tuning by checking for a CPU bottleneck. As shown in the following code, 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) * dToR);
                  y = sin((latitude+1) * dToR);
                  z = cos(longitude * dToR) * cos((latitude+1) * dToR);
                  glColor3f(x, y, z);
                  glColor3f(x, y, z);
                  }
            glEnd();
            }
      }
```

Using the Profiler

The program still renders less than 0.8 frames per second. Because eliminating all graphics output had almost no effect on performance, the program is clearly CPU-limited. Use the profiler to determine which function accounts for most of the execution time.

```
% cc -o perf -O -p perf.c -lGLU -lGL -lX11
% perf
% prof perf
```

```
------------------------------------------------------------
Profile listing generated Wed Jul 19 17:17:03 1995
    with:       prof perf
------------------------------------------------------------

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
    219   2.2s  R4000  R4010 100.0MHz   0     10.0ms     0(bytes)

Each sample covers 4 bytes for every 10.0ms (0.46% of 2.1900sec)
-----------------------------------------------------------------
-p[rocedures] using pc-sampling.
Sorted in descending order by the number of samples in each procedure.
Unexecuted procedures are excluded.
-----------------------------------------------------------------

samples   time(%)      cum time(%)      procedure (file)

    112   1.1s( 51.1)  1.1s( 51.1)      __sin
                                        (/usr/lib/libm.so:trig.s)
     29  0.29s( 13.2)  1.4s( 64.4)      Test (perf:perf.c)
     18  0.18s(  8.2)  1.6s( 72.6)      __cos (/usr/lib/libm.so:trig.s)
     16  0.16s(  7.3)  1.8s( 79.9)      Finish
                       (/usr/lib/libGLcore.so:../EXPRESS/gr2_context.c)
     15  0.15s(  6.8)  1.9s( 86.8)      __glexpim_Color3f
                       (/usr/lib/libGLcore.so:../EXPRESS/gr2_vapi.c)
     14  0.14s(  6.4)    2s( 93.2)      _BSD_getime
                       (/usr/lib/libc.so.1:BSD_getime.s)
      3  0.03s(  1.4)  2.1s( 94.5)      __glim_Finish
                       (/usr/lib/libGLcore.so:../soft/so_finish.c)
      3  0.03s(  1.4)  2.1s( 95.9)      _gettimeofday
                       (/usr/lib/libc.so.1:gettimeday.c)
      2  0.02s(  0.9)  2.1s( 96.8)      InitBenchmark (perf:perf.c)
      1  0.01s(  0.5)  2.1s( 97.3)      __glMakeIdentity
                       (/usr/lib/libGLcore.so:../soft/so_math.c)
      1  0.01s(  0.5)  2.1s( 97.7)      _ioctl
                       (/usr/lib/libc.so.1:ioctl.s)
      1  0.01s(  0.5)  2.1s( 98.2)       __glInitAccum64
                       (/usr/lib/libGLcore.so:../soft/so_accumop.c)
      1  0.01s(  0.5)  2.2s( 98.6)       _bzero
                       (/usr/lib/libc.so.1:bzero.s)
      1  0.01s(  0.5)  2.2s( 99.1)       GetClock (perf:perf.c)
      1  0.01s(  0.5)  2.2s( 99.5)       strncpy
                       (/usr/lib/libc.so.1:strncpy.c)
      1  0.01s(  0.5)  2.2s(100.0)      _select
                       (/usr/lib/libc.so.1:select.s)
```

```
    219   2.2s(100.0)  2.2s(100.0)        TOTAL
```

Almost 60% of the program's time for a single frame is spent computing trigonometric functions (__sin and __cos).

There are several ways to improve this situation. First, consider reducing the resolution of the quad strips that model the sphere. The current representation has over 60,000 quads, which is probably more than is needed for a high-quality image. After that, consider other changes like the following:

- 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); therefore, 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. The following code illustrates this tuning:

```
void
Test(void) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glCallList(1);
      }
....
void
RunTest(void){...
      glNewList(1, GL_COMPILE);
      for (latitude = -90; latitude < 90; ++latitude) {
            glBegin(GL_QUAD_STRIP);
            for (longitude = 0; longitude <= 360; ++longitude) {
                  GLfloat x, y, z;
                  x = sin(longitude * dToR) * cos(latitude * dToR);
                  y = sin(latitude * dToR);
                  z = cos(longitude * dToR) * cos(latitude * dToR);
                  glNormal3f(x, y, z);
```

```
                glVertex3f(x, y, z);
                x = sin(longitude * dToR) * cos((latitude+1) * dToR);
                y = sin((latitude+1) * dToR);
                z = cos(longitude * dToR) * cos((latitude+1) * dToR);
                glNormal3f(x, y, z);
                glVertex3f(x, y, z);
                }
        glEnd();
        }
glEndList();

printf("%.2f frames per second\n", Benchmark(Test));
}
```

This version of the program achieves a little less than 2.5 frames per second, a noticeable improvement.

When the **glClear()**, **glNormal3f()**, and **glVertex3f()** calls are again replaced with **glColor3f()**, the program runs at roughly 4 frames per second. This implies that the program is no longer CPU-limited. Therefore, 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 does not 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 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. Consider changing the following line

```
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, the example program 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, as shown in the following code. 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 rougher-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, a maximum of 200 frames per second, becomes apparent only because the program is single-buffered. If the program used double buffering, performance would not 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 and performance increases, 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). The following code illustrate this::

```
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.

System-Specific Tuning

This chapter first describes some general issues regarding system-specific tuning and 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 uses the following to topics:

- "Introduction to System-Specific Tuning"
- "Optimizing Performance on InfiniteReality Systems" on page 477
- "Optimizing Performance on Onyx4 and Silicon Graphics Prism Systems" on page 482

Some points are also described 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 man page for *`gfxinfo`* for more information. You can also call **glGetString()** with the `GL_RENDERER` argument. For information about the renderer strings for different systems, see the man page for **glGetString()**.

Introduction to System-Specific Tuning

Many of the performance tuning techniques described in the previous chapters (such as minimizing the number of state changes and disabling features that are not required) are good ideas, regardless of your platform. Other tuning techniques need to be customized for certain systems. For example, before you sort your database based on state changes, you need to determine which state changes are the most expensive for your target system.

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 must 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 must 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 that your program does not have to do this test each time it starts. You can find an example of how to measure the performance of particular OpenGL operations and to save the results using the `isfast` program from the OpenGL website.

Optimizing Performance on InfiniteReality Systems

This section describes 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.
- 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.

  On Infinite Reality systems, only the copy on the graphics pipe exists. If you run out of texture memory, OpenGL must save the texture that did not fit from the graphics pipe to the host, clean up texture memory, and then reload the texture. To avoid these multiple moves of the texture, always clean up textures you no longer need so that you do not deplete texture memory.

  This approach has the advantage of very fast texture loading because no host copy is made.
- To load a texture immediately, perform the following 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, perform the following steps:

  1. Disable texturing.
  2. Bind your texture.

3. Call **glTexImage*()**.

In this case, a copy of your texture is placed in main memory.

- Do not overflow texture memory; otherwise, texture swapping will occur.
- If you want to implement your own texture memory management policy, use subtexture loading. You have the following two options:
  - Allocate one large empty texture, call **glTexSubImage*()** to load it piecewise, and then use the texture matrix to select the relevant portion.
  - Allocate several textures, then fill them in by calling **glTexSubImage*()** as appropriate.

  For both options, it is important that after initial setup, you never create and destroy textures but reuse existing ones.
- 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 and 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.

Offscreen Rendering and Framebuffer Management

InfiniteReality systems support offscreen rendering through a combination of OpenGL features and extensions:

Pixel buffers	Pixel buffers (pbuffers) are offscreen pixel arrays that behave much like windows, except that they are invisible. See “SGIX_pbuffer—The Pixel Buffer Extension” on page 121.
Framebuffer configurations	Framebuffer configurations define color buffer depths, determine presence of Z buffers, and so on. See “Using Visuals and Framebuffer Configurations” on page 71.

Concurrent reads/writes	The function **glXMakeCurrentReadSGI()** allows you to read from one window or pbuffer while writing to another. See “SGI_make_current_read—The Make Current Read Extension” on page 114.

In addition, **glCopyTexImage*()** allows you to copy from a 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 will have less space in the framebuffer for pbuffers.
- A pbuffer is swappable (to avoid collisions with windows) but is 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.
- A pbuffer 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

The following items provide guidelines for 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, make the following calls:

  ```
  glLineWidth(width);
  glEnable(GL_LINE_SMOOTH);
  ```

  As a result of these calls, the line filter table is computed just once when line antialiasing is enabled. The table may be computed twice (once when antialiasing is enabled and again when the line width is changed) if you make the following calls:

  ```
  glEnable(GL_LINE_SMOOTH);
  glLineWidth(width);
  ```

  As a result, it may be best to disable a feature if you plan to change state and 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 the 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 a 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

The following are some 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 systems. It is less critical, but still recommended, on Onyx2 systems. The performance of the two systems 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" on page 305) 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.
- Render with exactly one thread per pipe.
- Use multiple OpenGL rendering contexts sparingly.

  Assuming no texture swapping, the rendering context-switching rate is about 60,000 calls per second. Therefore, each call to **glXMakeCurrent()** costs the equivalent of 100 textured triangles or 800 32-bit pixels.

Optimizing Performance on Onyx4 and Silicon Graphics Prism Systems

This section describes OpenGL performance optimizations for Onyx4 and Silicon Graphics Prism systems. Both Onyx4 and Silicon Graphics Prism systems use commodity graphics GPUs. Compared to older SGI graphics systems such as InfiniteReality and VPro, graphics hardware of this type differs substantially in features and in how to achieve peak performance (*fast paths*).

This section describes the following topics:

For a more complete discussion of performance issues, including higher-level issues such as multipipe scaling, see the document *Silicon Graphics UltimateVision Graphics Porting Guide*. You can also refer to the latest platform-specific documentation and release notes for your system, since additional performance optimizations and fast paths are ongoing.

Geometry Optimizations: Drawing Vertices

On older SGI graphics systems, immediate-mode rendering could usually reach peak performance of the geometry pipeline. However, the geometry pipeline capacity in Onyx4 and Silicon Graphics Prism GPUs greatly exceeds the available CPU-to-graphics bandwidth. The fastest paths for geometry on Onyx4 and Silicon Graphics Prism systems are either display lists or vertex buffer objects.

It is usually easiest to use display lists when porting older applications. When constructing display lists, a variety of optimizations are performed by the system. Some of these optimizations may be controlled by environment variables, as defined in platform-specific documentation.

Vertex buffer objects (using the ARB_vertex_buffer_object extension) are the preferred fast path when writing new code. When drawing indexed geometry, make sure to store both vertex array data and the array index data in buffer objects.

Drawing geometry using vertex buffer objects or display lists can be more than five times faster than immediate-mode rendering. The performance gain is typically larger on Onyx4 systems than on Silicon Graphics Prism systems.

Texturing Optimizations: Loading and Rendering Texture Images

The GPUs in Onyx4 and Silicon Graphics Prism systems support less texture memory than InfiniteReality systems. In addition, texture memory is shared with framebuffer and display list memory. This memory sharing may further reduce available available texture memory depending on the framebuffer configuration, use of pixel buffers, size of display lists, etc. However, you can reduce the texture memory requirements through the use of compressed texture formats.

Using OpenGL 1.3 core features and the EXT_texture_compression_s3tc extension, Onyx4 and Silicon Graphics Prism systems both support compressed texture formats. Compressed textures use approximately one-sixth of the space required for an equivalent uncompressed texture and require correspondingly less graphics memory bandwidth when rendering. Texture compression should be used whenever the resulting image quality loss is acceptable. When texture compression is not acceptable, use the fastest uncompressed texture formats, as described in the following section “Pixel Optimizations: Reading and Writing Pixel Data” on page 483.

In some cases, the graphics drivers may automatically compress textures by default. Refer to platform-specific documentation for more information about controlling this process.

Pixel Optimizations: Reading and Writing Pixel Data

When you use functions like **glDrawPixels()**, **glReadPixels()**, and **glTexImage2D()** to transfer pixel and uncompressed texture data between the CPU and graphics pipeline, it is much faster when you use pixel format and type combinations that are efficiently supported by the GPUs and drivers.

When reading and writing pixel data, the format `GL_RGBA` and type `GL_UNSIGNED_BYTE` are fastest. When reading and writing uncompressed texture images, the same format and type are fastest, as well as the internal texture format `GL_RGBA`. When writing format `GL_DEPTH_COMPONENT` (depth buffer data), the type `GL_UNSIGNED_SHORT` is fastest.

Other combinations of pixel format and type require additional conversion and packing/unpacking steps. Some additional format/type combinations may be optimized in the future; refer to the platform release notes for more information.

Differences Between Onyx4 and Silicon Graphics Prism Systems

In contrast to older SGI graphics systems, Onyx4 and Silicon Graphics Prism systems both use commodity graphics GPUs. The optimizations cited earlier in this section are applicable to commodity graphics GPUs on any system. However, the following differences between Onyx4 and Silicon Graphics Prism systems may affect performance:

- Onyx4 systems use MIPS CPUs while Silicon Graphics Prism systems use Intel Itanium CPUs. In general, Silicon Graphics Prism systems have higher CPU performance and greater memory bandwidth compared to Onyx4 systems. This affects compute-bound applications.
- Onyx4 systems run the IRIX operating system while Silicon Graphics Prism systems run Linux. The OpenGL and X feature sets of the two systems are very similar, and the operating system differences generally do not, in and of themselves, affect performance.
- Onyx4 systems use a PCI-X interface between the CPU and GPU while Silicon Graphics Prism systems use an AGP 8x interface. The AGP interface offers considerably higher bandwidth, which will improve performance for immediate-mode rendering, pixel and texture uploads and downloads, and other operations that must shift large amounts of data between the CPU and GPU.

  However, even on Silicon Graphics Prism systems, it is important to follow the advice cited earlier in this section regarding use of display lists and vertex buffer objects, efficient texture and pixel formats, etc. Neither the PCI-X interface nor the AGP interface is capable of feeding data to GPUs at a transfer rate equal to their processing rate. Hence, caching data on GPUs is critical to peak performance.
- Silicon Graphics Prism systems, a more recent product line, will have more opportunities for upgrades, resulting in greater performance and more OpenGL features, to both CPUs and GPUs.

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 17, "Tuning Graphics Applications: Examples."

```
/*************************************************************************
 * perf - framework for measuring performance of an OpenGL operation
 *
 * Compile with: cc -o perf -O perf.c -lGL -lX11
 *
*************************************************************************/


#include <GL/glx.h>
#include <X11/keysym.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <sys/time.h>


char* ApplicationName;
double Overhead = 0.0;
int VisualAttributes[] = { GLX_RGBA, None };
int WindowWidth;
int WindowHeight;


/*************************************************************************
 * GetClock - get current time (expressed in seconds)
*************************************************************************/
double
GetClock(void) {
        struct timeval t;

        gettimeofday(&t);
        return (double) t.tv_sec + (double) t.tv_usec * 1E-6;
        }
```

```
/*************************************************************************
 * ChooseRunTime - select an appropriate runtime for benchmarking
*************************************************************************/
double
ChooseRunTime(void) {
        double start;
        double finish;
        double runTime;

        start = GetClock();

        /* Wait for next tick: */
        while ((finish = GetClock()) == start)
                ;

        /* Run for 100 ticks, clamped to [0.5 sec, 5.0 sec]: */
        runTime = 100.0 * (finish - start);
        if (runTime < 0.5)
                runTime = 0.5;
        else if (runTime > 5.0)
                runTime = 5.0;

        return runTime;
        }

/*************************************************************************
 * FinishDrawing - wait for the graphics pipe to go idle
 *
 * This is needed to make sure we're not including time from some
 * previous uncompleted operation in our measurements.  (It's not
 * foolproof, since we can't eliminate context switches, but we can
 * assume our caller has taken care of that problem.)
*************************************************************************/
void
FinishDrawing(void) {
        glFinish();
        }

/*************************************************************************
 * WaitForTick - wait for beginning of next system clock tick; return
 * the time
*************************************************************************/
```

```
double
WaitForTick(void) {
        double start;
        double current;

        start = GetClock();

        /* Wait for next tick: */
        while ((current = GetClock()) == start)
                ;

        /* Start timing: */
        return current;
        }

/*********************************************************************
 * InitBenchmark - measure benchmarking overhead
 *
 * This should be done once before each risky change in the
 * benchmarking environment.  A ``risky'' change is one that might
 * reasonably be expected to affect benchmarking overhead.  (For
 * example, changing from a direct rendering context to an indirect
 * rendering context.)  If all measurements are being made on a single
 * rendering context, one call should suffice.
*********************************************************************/
void
InitBenchmark(void) {
        double runTime;
        long reps;
        double start;
        double finish;
        double current;

        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /* Wait for the pipe to clear: */
        FinishDrawing();

        /* Measure approximate overhead for finalization and timing
         * routines
         */
        reps = 0;
        start = WaitForTick();
        finish = start + runTime;
```

```
        do {
                FinishDrawing();
                ++reps;
                } while ((current = GetClock()) < finish);

        /* Save the overhead for use by Benchmark(): */
        Overhead = (current - start) / (double) reps;
        }

/*****************************************************************
 * Benchmark - measure number of caller's operations performed per
 * second.
 * Assumes InitBenchmark() has been called previously, to initialize
 * the estimate for timing overhead.
*****************************************************************/
double
Benchmark(void (*operation)(void)) {
        double runTime;
        long reps;
        long newReps;
        long i;
        double start;
        double current;

        if (!operation)
                return 0.0;

        /* Select a run time appropriate for our timer resolution: */
        runTime = ChooseRunTime();

        /*
         * Measure successively larger batches of operations until we
         * find one that's long enough to meet our runtime target:
         */
        reps = 1;
        for (;;) {
                /* Run a batch: */
                FinishDrawing();
                start = WaitForTick();
                for (i = reps; i > 0; --i)
                        (*operation)();
                FinishDrawing();
```

```
                /* If we reached our target, bail out of the loop: */
                current = GetClock();
                if (current >= start + runTime + Overhead)
                        break;

                /*
                 * Otherwise, increase the rep count and try to reach
                 * the target on the next attempt:
                 */
                if (current > start)
                        newReps = reps *
                                (0.5 + runTime / (current - start -
                                                        Overhead));
                else
                        newReps = reps * 2;
                if (newReps == reps)
                        reps += 1;
                else
                        reps = newReps;
                }

        /* Subtract overhead and return the final operation rate: */
        return (double) reps / (current - start - Overhead);
        }

/*************************************************************************
 * Test - the operation to be measured
 *
 * Will be run several times in order to generate a reasonably accurate
 * result.
*************************************************************************/
void
Test(void) {
        /* Replace this code with the operation you want to measure: */
        glColor3f(1.0, 1.0, 0.0);
        glRecti(0, 0, 32, 32);
        }

/*************************************************************************
 * RunTest - initialize the rendering context and run the test
*************************************************************************/
void
RunTest(void) {
        if (Overhead == 0.0)
                InitBenchmark();
```

```
        /* Replace this sample with initialization for your test: */

        glClearColor(0.5, 0.5, 0.5, 1.0);
        glClear(GL_COLOR_BUFFER_BIT);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0.0, WindowWidth, 0.0, WindowHeight, -1.0, 1.0);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        printf("%.2f operations per second\n", Benchmark(Test));
        }

/*****************************************************************
 * ProcessEvents - handle X11 events directed to our window
 *
 * Run the measurement each time we receive an expose event.
 * Exit when we receive a keypress of the Escape key.
 * Adjust the viewport and projection transformations when the window
 * changes size.
*****************************************************************/
void
ProcessEvents(Display* dpy) {
        XEvent event;
        Bool redraw = 0;

        do {
                char buf[31];
                KeySym keysym;

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

```
                        case KeyPress:
                                (void) XLookupString(&event.xkey, buf,
                                        sizeof(buf), &keysym, NULL);
                                switch (keysym) {
                                        case XK_Escape:
                                                exit(EXIT_SUCCESS);
                                        default:
                                                break;
                                        }
                                break;
                        default:
                                break;
                        }
                } while (XPending(dpy));

        if (redraw) RunTest();
        }

/*****************************************************************
 * Error - print an error message, then exit
*****************************************************************/
void
Error(const char* format, ...) {
        va_list args;

        fprintf(stderr, "%s:  ", ApplicationName);

        va_start(args, format);
        vfprintf(stderr, format, args);
        va_end(args);

        exit(EXIT_FAILURE);
        }

/*****************************************************************
 * main - create window and context, then pass control to ProcessEvents
*****************************************************************/
int
main(int argc, char* argv[]) {
        Display *dpy;
        XVisualInfo *vi;
        XSetWindowAttributes swa;
        Window win;
        GLXContext cx;
```

```
ApplicationName = argv[0];

/* Get a connection: */
dpy = XOpenDisplay(NULL);
if (!dpy) Error("can't open display");

/* Get an appropriate visual: */
vi = glXChooseVisual(dpy, DefaultScreen(dpy),VisualAttributes);
if (!vi) Error("no suitable visual");

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

/* Create a color map: */
swa.colormap = XCreateColormap(dpy, RootWindow(dpy,
                          vi->screen), vi->visual, AllocNone);

/* Create a window: */
swa.border_pixel = 0;
swa.event_mask = ExposureMask | StructureNotifyMask |
                                                  KeyPressMask;
win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0,
               300, 300, 0,vi->depth, InputOutput, vi->visual,
               CWBorderPixel|CWColormap|CWEventMask, &swa);
XStoreName(dpy, win, "perf");
XMapWindow(dpy, win);

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

/* Handle events: */
while (1) ProcessEvents(dpy);
}
```

Benchmarking Libraries: `libpdb` and `libisfast`

When optimizing an OpenGL application, there are two problems you need to address:

- When you are writing an OpenGL application, it is 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 run time.

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. The following are examples:

- A machine might have hardware support for depth buffering but only for some comparison functions.
- Depth buffering might be fast only as long as stencilling is not enabled.
- Depth buffering might be fast when drawing to a window but slow when drawing to a pixmap.

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 run-time 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 OpenGL Performer.

Performance measurement can be tricky, as indicated by the following considerations:

- You need to handle the cases when you are displaying over a network as well as locally.
- You must flush the graphics pipeline properly and account 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 will not 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 single-processor and multiprocessor systems, and so on.

Libraries for Benchmarking

This appendix describes two libraries that can help with all of the tasks just mentioned:

libpdb	Performance database 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 cannot substitute for comprehensive benchmarking and performance analysis and do not replace more sophisticated tools (like OpenGL Performer and Open Inventor) that optimize application performance in a variety of ways. However, they can handle simple tasks easily.

Using libpdb

Library `libpdb` provides the following 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 bit mask 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 B-1.

Table B-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`	Routine **pdbOpen()** was not invoked before calling one of the other `libpdb` routines.
`PDB_ALREADY_OPEN`	Routine **pdbOpen()** was called while the database is still open (for example, before **pdbClose()** is invoked).

Every program must call **pdbOpen()** before using the database and **pdbClose()** when the database is no longer needed. Routine **pdbOpen()** opens the database file (stored in `$HOME/.pdb2` on IRIX and Linux systems) and reads all the performance measurements into main memory. Routine **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()**. The following are sample calls to the routines:

```
pdbStatusT pdbOpen(void);
pdbStatusT pdbClose(void);
```

Routine **pdbOpen()** returns the following:

- 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()**

Routine **pdbClose()** returns the following:

- 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. Routine **pdbReadRate()**, which is used for this purpose, has the following format:

```
pdbStatusT pdbReadRate (const char* machineName,const char* appName,
        const char* benchmarkName,const char* versionString, double* rate)
```

The variable items are defined as follows:

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 pdbReadRate()

The following example illustrates the use of **pdbReadRate()**:

```
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 format, to make a measurement:

```
pdbStatusT pdbMeasureRate (pdbCallbackT initialize, pdbCallbackT operation,
                           pdbCallbackT finalize, int calibrate, double* rate)
```

The variable items are defined as follows:

initialize A pointer to the initialization function. The initialization function is run before each set of operations. For OpenGL performance measurement, it is 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. The value 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 will want to make sure that any global state needed by the operation is set up before calling the operation function so that you do not 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 preceding example, **glFinish()** ensures that the graphics pipeline is idle. The value of *finalize* 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. Routine **pdbMeasureRate()** attempts to compute a number of repetitions that results in a run time of about one second. (Calibration requires an additional second.) It is reasonably careful about timekeeping on systems with low-resolution clocks.

Routine **pdbMeasureRate()** always returns `PDB_NO_ERROR`.

Example for pdbMeasureRate()

The following example illustrates the use of **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 format:

```
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.

Routine **pdbWriteRate()** returns the following:

- `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
- `PDB_NOT_OPEN` if the database is not open

When **pdbWriteRate()** is called, the in-memory copy of the performance database is marked "dirty." Routine **pdbClose()** takes note of this and writes the database back to disk.

Example for pdbWriteRate()

The following example illustrates the use of **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. Library libisfast is based on purely subjective performance criteria. If they are 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 libisfast yield more useful results when the target machine is being accessed over a network.

The application must initialize libisfast before performing any performance measurements and clean up after the measurements are finished. On X11 systems, initialize libisfast by making the following call:

```
int IsFastXOpenDisplay(const char* displayName);
```

Perform cleanup by making the following call:

```
void IsFastXCloseDisplay(void);
```

The following are other `libisfast` routines to use:

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 half as fast as triangles without depth buffering:

```
int DepthBufferingIsFast(void);
```

ImmediateModeIsFast() Returns nonzero if immediate-mode triangles can be drawn at least 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 half as fast as non-stencilled triangles:

```
int StencillingIsFast(void);
```

TextureMappingIsFast() Returns nonzero if texture-mapped triangles can be drawn at least 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. Doing so allows yout to 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.

System Support for OpenGL Versions and Extensions

Using the following topics, this appendix lists the OpenGL core versions and extensions that are supported on the various Silicon Graphics visualization systems:

OpenGL Core Versions

Table C-1 shows the support for OpenGL core versions by system.

Table C-1 Support for OpenGL Core Versions

OpenGL and GLX Version	Visualization System
OpenGL 1.1 and GLX 1.3	InfiniteReality Also supports many EXT and SGI extensions.
OpenGL 1.2 and GLX 1.3	Fuel, Tezro, and InfinitePerformance systems using VPro graphics Also supports many EXT and SGI extensions.
OpenGL 1.3 and GLX 1.3	Silicon Graphics Onyx4 UltimateVision and Silicon Graphics Prism systems using commodity GPUs from ATI Technologies Also supports many standard ARB and ATI vendor-specific extensions. Some EXT and SGI extensions common to VPro and InfiniteReality are also supported.

In general, the sets of extensions supported by traditional Silicon Graphics systems and by the newer GPU-based Silicon Graphics systems are largely disjoint. However, by writing appropriate user-defined vertex and fragment programs, many of the vertex

processing and rasterization features introduced with older EXT and SGI extensions can be emulated.

OpenGL Extensions

Table C-2 lists the supported OpenGL extensions. Note that while the list is comprehensive, this guide only describes those extensions that are either available or scheduled to be available on more than one platform.

Table C-2 OpenGL Extensions on Different Silicon Graphics Systems

Extension	VPro/ InfinitePerformance	InfiniteReality	UltimateVision/ Prism
ARB_depth_texture			X
ARB_fragment_program			X
ARB_imaging	X		
ARB_multisample			X
ARB_multitexture			X
ARB_point_parameters			X
ARB_shadow			X
ARB_shadow_ambient			X
ARB_texture_border_clamp			X
ARB_texture_compression			X
ARB_texture_cube_map			X
ARB_texture_env_add			X
ARB_texture_env_combine			X
ARB_texture_env_crossbar			X
ARB_texture_env_dot3			X
ARB_texture_mirrored_repeat			X
ARB_transpose_matrix			X

Table C-2 OpenGL Extensions on Different Silicon Graphics Systems **(continued)**

Extension	VPro/ InfinitePerformance	InfiniteReality	UltimateVision/ Prism
ARB_vertex_blend			X[a]
ARB_vertex_buffer_object			X[b]
ARB_vertex_program			X
ARB_window_pos			X
ATIX_texture_env_combine3			X
ATIX_texture_env_route			X
ATIX_vertex_shader_output_point_size			X
ATI_draw_buffers			X
ATI_element_array			X
ATI_envmap_bumpmap			X
ATI_fragment_shader			X
ATI_map_object_buffer			X
ATI_separate_stencil			X
ATI_texture_env_combine3			X
ATI_texture_float			X
ATI_texture_mirror_once			X
ATI_vertex_array_object			X
ATI_vertex_attrib_array_object			X
ATI_vertex_streams			X
EXT_abgr	X	X	X
EXT_bgra			X
EXT_blend_color	X	X	X
EXT_blend_func_separate			X

Table C-2 OpenGL Extensions on Different Silicon Graphics Systems **(continued)**

Extension	VPro/ InfinitePerformance	InfiniteReality	UltimateVision/ Prism
EXT_blend_logic_op	X	X	X[c]
EXT_blend_minmax	X	X	X
EXT_blend_subtract	X	X	X
EXT_clip_volume_hint			X
EXT_compiled_vertex_array			X
EXT_convolution	X	X	
EXT_copy_texture	X		X[d]
EXT_draw_range_elements			X
EXT_fog_coord			X
EXT_histogram	X	X	
EXT_multi_draw_arrays			X
EXT_packed_pixels	X	X	X
EXT_point_parameters			X
EXT_polygon_offset	X		X
EXT_rescale_normal			X
EXT_secondary_color			X
EXT_separate_specular_color			X
EXT_stencil_wrap			X
EXT_subtexture	X		X[e]
EXT_texgen_reflection			X
EXT_texture	X		X[f]
EXT_texture3D	X	X	X
EXT_texture_compression_s3tc			X

Table C-2 OpenGL Extensions on Different Silicon Graphics Systems **(continued)**

Extension	VPro/ InfinitePerformance	InfiniteReality	UltimateVision/ Prism
EXT_texture_cube_map			X
EXT_texture_edge_clamp			X
EXT_texture_env_add	X		X
EXT_texture_env_combine			X
EXT_texture_env_dot3			X
EXT_texture_filter_anisotropic			X
EXT_texture_lod_bias			X
EXT_texture_object	X		X
EXT_texture_rectangle			X
EXT_vertex_array	X		X
EXT_vertex_shader			X
HP_occlusion_test			X
INGR_interlace_read	X		
NV_blend_square			X
NV_occlusion_query			X
NV_point_sprite			X
NV_texgen_reflection			X
S3_s3tc			X
SGI_color_matrix	X	X	X
SGI_color_table	X	X	
SGI_texture_color_table	X	X	
SGIS_detail_texture	X	X	
SGIS_fog_function	X	X	

Table C-2 OpenGL Extensions on Different Silicon Graphics Systems **(continued)**

Extension	VPro/ InfinitePerformance	InfiniteReality	UltimateVision/ Prism
SGIS_generate_mipmap			X
SGIS_multisample		X	
SGIS_multitexture			X[g]
SGIS_pixel_texture	X		
SGIS_point_line_texgen		X	
SGIS_point_parameters		X	
SGIS_sharpen_texture		X	
SGIS_texture_LOD		X	
SGIS_texture_border_clamp	X		X
SGIS_texture_color_mask	X		
SGIS_texture_edge_clamp	X	X	X
SGIS_texture_filter4		X	
SGIS_texture_lod	X		X
SGIS_texture_select		X	
SGIX_async	X		
SGIX_async_pixel	X		
SGIX_blend_alpha_minmax	X		
SGIX_calligraphic_fragment		X	
SGIX_clipmap		X	
SGIX_convolution_accuracy	X		
SGIX_depth_texture		X	
SGIX_flush_raster		X	
SGIX_fog_offset	X	X	

Table C-2 OpenGL Extensions on Different Silicon Graphics Systems **(continued)**

Extension	VPro/ InfinitePerformance	InfiniteReality	UltimateVision/ Prism
SGIX_fragment_lighting	X		
SGIX_instruments		X	
SGIX_interlace	X	X	
SGIX_ir_instrument1		X	
SGIX_list_priority	X	X	
SGIX_reference_plane		X	
SGIX_resample	X		
SGIX_scalebias_hint	X		
SGIX_shadow		X	
SGIX_shadow_ambient		X	
SGIX_sprite		X	
SGIX_subsample	X		
SGIX_texture_add_env		X	
SGIX_texture_coordinate_clamp	X		
SGIX_texture_lod_bias	X	X	
SGIX_texture_scale_bias	X	X	
SGIX_vertex_preclip	X		
SUN_multi_draw_arrays			X[h]

a. Silicon Graphics Prism systems only
b. Silicon Graphics Prism systems only
c. Silicon Graphics Onyx4 UltimateVision systems only
d. Silicon Graphics Onyx4 UltimateVision systems only
e. Silicon Graphics Onyx4 UltimateVision systems only
f. Silicon Graphics Onyx4 UltimateVision systems only
g. Silicon Graphics Prism systems only
h. Silicon Graphics Prism systems only

GLX Extensions

Table C-3 lists the GLX extensions supported on Silicon Graphics systems.

Table C-3 GLX Extensions on Different Silicon Graphics Systems

Extension	VPro/ InfinitePerformance	InfiniteReality	UltimateVision/ Prism
GLX_ARB_get_proc_address			X
GLX_ARB_multisample			X
GLX_SGIS_multisample		X	
GLX_EXT_import_context	X	X	X
GLX_EXT_visual_info	X	X	X
GLX_EXT_visual_rating	X	X	X
GLX_SGIX_fbconfig	X	X	X[a]
GLX_SGIX_pbuffer	X	X	X[b]
GLX_SGIX_hyperpipe	X	X	X
GLX_SGIX_swap_barrier	X	X	X [c]
GLX_SGIX_swap_group	X	X	X
GLX_SGI_swap_control	X	X	
GLX_SGI_make_current_read	X	X	X
GLX_SGI_video_sync	X	X	X[d]
GLX_SGIX_video_resize		X	

a. Silicon Graphics Onyx4 UltimateVision systems only. In new code, use GLX core features to access FBConfigs and pixel buffers instead of this extension.

b. Silicon Graphics Onyx4 UltimateVision systems only. In new code, use GLX core features to access FBConfigs and pixel buffers instead of this extension.

c. Silicon Graphics Prism systems only.

d. Silicon Graphics Prism systems only.

XFree86 Configuration Specifics

Silicon Graphics Prism and Onyx4 systems require a number of system-specific X configuration settings for various configurations. Using the following topics, this appendix provides information about customizing the XF86Config file for Silicon Graphics Prism systems:

For Onyx4 systems, refer to the *Silicon Graphics Onyx4 UltimateVision User's Guide* and the *Silicon Graphics UltimateVision Graphics Porting Guide*. In general, for ongoing enhancements in X configuration, refer to the most recent user's guide for your platform.

Configuring a System for Stereo

This section describes how to configure a system to display stereo images.

Stereo sync is supported only on systems using ImageSync boards.

Note: Simultaneously running stereo and full-scene antialiasing can require more graphics-card memory than is available, and thus may not always work correctly.

---

1. Create a copy of the `XF86Config` file to be customized for stereo:

   ```
   # cp /etc/X11/XF86Config /etc/X11/XF86Config.Stereo
   ```

2. Edit the `XF86Config.Stereo` file to include the following line at the end of each "Device" section:

   ```
   Option "Stereo"           "1"
   Option "StereoSyncEnable" "1"
   ```

   (see the "Example "Device" Section for Stereo" on page 513).

3. Edit the `XF86Config.Stereo` file to include the appropriate stereo modes in the "Monitor" section:

   a. Create an appropriate mode (see "Sample Stereo Mode Entries" on page 513).

   b. Add that mode to the "Monitor" section of your `XF86Config.Stereo` file (see the "Example "Monitor" Section for Stereo" on page 514).

   Note: "Mode" and "Modeline" are two alternative formats used to present the same information.

   ---

4. Ensure that the monitor supports the high horizontal sync rate setting. Refer to the documentation for the monitor to determine the horizontal sync rate. Modify the `HorizSync` setting in the "Monitor" section of the `XF86Config.Stereo` file. For example:

   ```
   HorizSync   22-105
   ```

5. Modify the "Screen" section so that you use the appropriate mode setting. For example:

Modes "1280x1024@96" (see the "Example "Screen" Section for Stereo" on page 514).

6. Create a new /etc/X11/xdm/`Xservers.Stereo` file containing the following line:

   ```
   :0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config.Stereo
   ```

7. Edit the /etc/X11/xdm/`xdm-config` file to point to the new X servers file:

   Replace the line:

   ```
   DisplayManager.servers: /etc/X11/xdm/Xservers
   ```

   with:

   ```
   DisplayManager.servers: /etc/X11/xdm/Xservers.Stereo
   ```

8. Save the file and reboot the system to restart graphics in stereo mode:

Note that a stereo sync signal will not be present until you run a stereo application. One such application is `ivview`. If your system has `ivview` installed, you can use it to test the stereo configuration by running:
`ivview /usr/share/data/models/X29.iv`
and right click to activate the stereo setting on the preferences panel.

Example “Device” Section for Stereo

```
Section "Device"
    Identifier  "SGI SG-0"
    Driver      "fglrx"
    BusId       "PCI:2:0:0"
# === QBS Management ===
    Option "Stereo"           "1"
    Option "StereoSyncEnable" "1"
EndSection
```

Sample Stereo Mode Entries

```
Modeline "1024x768@96" 103.5  1024 1050 1154 1336  768 771 774 807
Modeline "1280x1024@96" 163.28  1280 1300 1460 1600  1024 1027 1033 1063
Modeline "1024x768@100" 113.309 1024 1096 1208 1392 768 769 772 814
Modeline "1024x768@120" 139.054 1024 1104 1216 1408 768 769 772 823 +hsync +vsync
Modeline "1280x1024@100" 190.960 1280 1376 1520 1760 1024 1025 1028 1085 +hsync +vsync
Mode "1280x1024_96s_mirage"
    DotClock       152.928
    HTimings       1280 1330 1390 1500
    VTimings       1024 1026 1030 1062
```

```
EndMode
```

Example “Monitor” Section for Stereo

```
Section "Monitor"
    Identifier  "Stereo Monitor"
    HorizSync   30-96      # multisync
    VertRefresh 50-160     # multisync
    Modeline "1024x768@96" 103.5  1024 1050 1154 1336  768 771 774 807
EndSection
```

Example “Screen” Section for Stereo

```
Section "Screen"
    Identifier          "Screen SG-0"
    Device              "SGI SG-0"
    Monitor             "Stereo Monitor"
    DefaultDepth         24
    SubSection          "Display"
        Depth   24
        Modes   "1280x1024@96"
    EndSubSection
EndSection
```

Configuring a System for Full-Scene Antialiasing

This section describes how to configure a system for global or per-window full-scene antialiasing.

Note: Simultaneously running stereo and full-scene antialiasing can require more graphics-card memory than is available, and thus may not work correctly.

---

1. Create a copy of the `XF86Config` file to be customized for full-scene antialiasing:

   ```
   # cp /etc/X11/XF86Config /etc/X11/XF86Config.AntiAlias
   ```

   Note: Automatically generated `XF86Config` files should contain the customized multi-sample positions shown in on page 516. If these values are not already present, adding them will significantly improve the quality of your output.

   ---

2. Edit the `XF86Config.AntiAlias` file to include the following line at the end of each "Device" section:

   `Option "FSAAScale" "`*X*`"`

   where *X* is 1, 2, 4, or 6 (see the example "Device" section on page 516).

   Note: Per-window full-scene antialiasing is accomplished by setting "FSAAScale" to 1. The antialiasing level may then be set by the appropriate selection of visuals. Global antialiasing is accomplished by setting "FSAAScale" to 2, 4, or 6. In this case, the setting will apply to all OpenGL windows, regardless of the visual being displayed.

   ---

3. Create a new `/etc/X11/xdm/Xservers.AntiAlias` file containing the following (all on one line):

```
:0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config.AntiAlias
```

4. Edit the `/etc/X11/xdm/xdm-config` file to point to the new X servers file:

   Replace the line:

   ```
   DisplayManager.servers: /etc/X11/xdm/Xservers
   ```

   with:

   ```
   DisplayManager.servers: /etc/X11/xdm/Xservers.AntiAlias
   ```

5. Stop the system graphics from the X-terminal:

   # <CTRL> <ALT> <BKSPC>

6. Restart graphics:

   # /usr/bin/X11/startx

Example “Device” Section for Full-Scene Antialiasing

```
Section "Device"
    Identifier  "SGI SG-0"
    Driver      "fglrx"
    BusId       "PCI:2:0:0"
# === FSAA Management ===
    Option "FSAAScale"                  "1"
    Option "FSAADisableGamma"           "no"
    Option "FSAACustomizeMSPos"         "yes"
    Option "FSAAMSPosX0"                "0.250000"
    Option "FSAAMSPosY0"                "0.416666"
    Option "FSAAMSPosX1"                "0.083333"
    Option "FSAAMSPosY1"                "0.083333"
    Option "FSAAMSPosX2"                "0.416666"
    Option "FSAAMSPosY2"                "0.750000"
    Option "FSAAMSPosX3"                "0.750000"
    Option "FSAAMSPosY3"                "0.916666"
    Option "FSAAMSPosX4"                "0.583333"
    Option "FSAAMSPosY4"                "0.250000"
    Option "FSAAMSPosX5"                "0.916666"
    Option "FSAAMSPosY5"                "0.583333"
EndSection
```

Configuring a System for Dual-Channel Operation

To configure a system for dual-channel operation, follow the steps in this section.

Note: If any pipes managed by an X server have their second channel enabled, then every pipe managed by that X server must have its second channel enabled.

Note: Both channels on a pipe must have the same display resolution.

1. Create a copy of the XF86Config file to be customized for dual-channel operation:

   ```
   # cp /etc/X11/XF86Config /etc/X11/XF86Config.DualChannel
   ```

2. Edit the `XF86Config.DualChannel` file to include the following line at the end of each "Device" section:

   ```
   Option "DesktopSetup" mode
   ```

   where *mode* is one of the following:

   ```
   "0x00000100" [this mode clones the managed area]
   "0x00000200" [this mode scales the managed area by 2 horizontally]
   "0x00000300" [this mode scales the managed area by 2 vertically]
   ```

   (see the example "Device" section on page 518).

   Note: All pipes managed by the same X server must be set to the same mode.

3. When using monitors or monitor cables which do not conform to the VESA Display Data Channel (DDC) standard, append the following entry in the "Device" section to enable proper display configuration:

   ```
   Option "NoDDC" "on"
   ```

4. Create a new `/etc/X11/xdm/Xservers.DualChannel` file containing the following line:

```
:0 secure /usr/bin/X11/X :0 -xf86config /etc/X11/XF86Config.DualChannel
```

5. Edit the `/etc/X11/xdm/xdm-config` file to point to the new X servers file:

   Replace the line:

   ```
   DisplayManager.servers: /etc/X11/xdm/Xservers
   ```

with:

```
DisplayManager.servers: /etc/X11/xdm/Xservers.DualChannel
```

6. Stop the system graphics from the X-terminal:

   # <CTRL> <ALT> <BKSPC>

7. Restart graphics:

   # /usr/bin/X11/startx

Example “Device” Section for Dual Channel

```
Section "Device"
    Identifier  "SGI SG-0"
    Driver      "fglrx"
    BusId       "PCI:2:0:0"
    Option      "DesktopSetup" "0x00000200"
EndSection
```

Enabling Overlay Planes

To enable overlay planes, follow these steps:

1. Edit the `/etc/X11/XF86Config` file to include the following line in each “Device” section for which you want overlay planes enabled:

   ```
   Option "OpenGLOverlay"  "On"
   ```

2. Log out from the desktop, then log back on.

Example “Device” Section to Enable Overlay Planes

```
Section "Device"
    Identifier  "SGI SG-0"
    Driver      "fglrx"
    BusId       "PCI:2:0:0"
    Option      "OpenGLOverlay" "On"
EndSection
```

Configuring a System for External Genlock or Framelock

External genlock and framelock may be used on systems with at least one optional ImageSync board.

To configure your system to receive an external genlock or framelock signal you must run the `setmon` command with the appropriate options.

Before running setmon, use `printenv DISPLAY` to ensure that the `DISPLAY` environment variable is set to the local system (for example, :0.0). If it is not, use `setenv DISPLAY :0.0` to change it (substituting other numbers for :0.0 if appropriate).

To set the system for genlock, execute the following command:

`# setmon -p`*`pipenumber`* `-g` *`graphicsformat`*

where *pipenumber* is the pipe to which this setting should be applied, and *graphicsformat* is one of the timings (modes) listed in the "Monitor" section of the `/etc/X11/XF86Config` file.

To set the system for framelock, execute the following command:

`# setmon -p`*`pipenumber`* `-L`*`videoformat`* *`graphicsformat`*

where *pipenumber* is the pipe to which this setting should be applied, *videoformat* is the input video format to be used as a framelock source, and *graphicsformat* is one of the framelock-certified timings (modes) listed in the "Monitor" section of the `/etc/X11/XF86Config` file that is compatible with the chosen input video format (Table D-1 on page 520 provides a list of compatible formats).

Note: The default behavior of `setmon` is to load the new format for the current session only and to prompt for input to determine if the format should be saved as the default. To save the new format as the default you must be logged in as root.

For more information about the `setmon` command, see the `setmon` man page (`man setmon`).

Note: Framelock-certified timings will include an "f" appended to their name (that is, "1280x1024_5994f" is certified for NTSC (525 line) video timing).

Table D-1 Input Video Formats (Framelock)

Input Video Format (Framelock Source)	Format Name	Compatible Graphics Formats
525 line at 59.94Hz (NTSC)	525 (or use the alias NTSC)	1280x1024_5994f 1920x1154_5994f
625 line at 50Hz (PAL)	625 (or use the alias PAL)	1280x1024_50f 1920x1154_50f
720-line progressive-scan at 59.94Hz	720p_5994	1920x1154_5994f
720-line progressive-scan at 60Hz	720p_60	1280x1024_60f 1920x1154_60f 1920x1200_60f
1080-line progressive-scan at 25Hz	1080p_25	1280x1024_50f 1920x1154_50f
1080-line interlaced at 25Hz	1080i_25	1280x1024_50f 1920x1154_50f
1080-line progressive-scan at 29.97Hz	1080p_2997	1920x1154_5994f
1080-line interlaced at 29.97Hz	1080i_2997	1920x1154_5994f
1080-line progressive-scan at 30Hz	1080p_30	1280x1024_60f 1920x1154_60f 1920x1200_60f
1080-line interlaced at 30Hz	1080i_30	1280x1024_60f 1920x1154_60f 1920x1200_60f

Configuring Monitor Positions

When an X-Server is managing multiple monitors, it needs to know their relative positions in order to properly handle cursor cross-over locations.

The monitor positions are specified in the “ServerLayout” section of the `/etc/X11/XF86Config` file as follows:

Each screen is listed, followed by a list of the screens above, below, to the left, and to the right of it (in that order). Figure D-1 and the following subsection show an example of four monitors arranged in a line.

Programs started by clicking on an icon appear on the screen from which you invoked them. Note that once a program has been launched, it is not possible to move it from one screen to another.

Figure D-1 Four Monitors in a Line

Example “ServerLayout” Section for Four Monitors in a Line

```
Section "ServerLayout"
    Identifier  "Four-in-a-Line"
    Screen "Screen SG-0"        ""      ""      ""      "Screen SG-1"
    Screen "Screen SG-1"        ""      ""      "Screen SG-0"   "Screen SG-2"
    Screen "Screen SG-2"        ""      ""      "Screen SG-1"   "Screen SG-3"
    Screen "Screen SG-3"        ""      ""      "Screen SG-2"   ""
    InputDevice "Mouse1" "CorePointer"
    InputDevice "Keyboard1" "CoreKeyboard"
EndSection
```

Figure D-2 and the subsection following it show an example of four monitors arranged in a square.

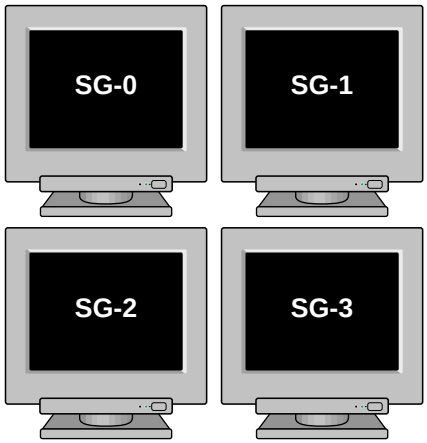


Figure D-2 Four Monitors in a Square

Example “ServerLayout” Section for Four Monitors in a Square

```
Section "ServerLayout"
    Identifier  "Four-in-a-Square"
    Screen "Screen SG-0"        ""      "Screen SG-2"      ""      "Screen SG-1"
    Screen "Screen SG-1"        ""      "Screen SG-3"      "Screen SG-0"   ""
    Screen "Screen SG-2"        "Screen SG-0"      ""      ""   "Screen SG-3"
    Screen "Screen SG-3"        "Screen SG-1"      ""      "Screen SG-2"   ""
    InputDevice "Mouse1" "CorePointer"
    InputDevice "Keyboard1" "CoreKeyboard"
EndSection
```

Configuring Monitor Types

The system graphics cards support both analog and digital monitors. The type of monitor connected to each graphics card is specified in the "Device" sections of the `/etc/X11/XF86Config` file.

Table D-2 lists the allowable options for the MonitorLayout line. If the line is not present, both channels default to `AUTO`.

Table D-2 Options for Monitor Layout

Monitor Type	Meaning
`AUTO`	Automatically select monitor type (default)
`TMDS`	Digital monitor
`CRT`	Analog monitor
`NONE`	No monitor

The format is:

`Option "MonitorLayout" "`*`channel1type`*`, `*`channel2type`*`"`

where *channel1type* is the type (`AUTO`, `TMDS`, `CRT` or `NONE`) of monitor attached to channel 1 (the left DVI-I connector) for this pipe, and
channel2type is the type (`AUTO`, `TMDS`, `CRT` or `NONE`) of monitor attached to channel 2 (the right DVI-I connector) for this pipe.

Example "Device" Section for Use With Two Analog Monitors

```
Section "Device"
    Identifier  "SGI SG-0"
    Driver      "fglrx"
    BusId       "PCI:2:0:0"
    Option      "MonitorLayout" "CRT, CRT"
EndSection
```

Configuring a System for Multiple X Servers

Multiple X servers allows specific subsets of the keyboards, pointing devices, and monitors attached to a Silicon Graphics Prism system to each be managed by a different X server.

Note: The use of multiple X servers requires SGI ProPack 3, Service Pack 4 or a later release of the software.

This section describes a relatively simple configuration. Much more complex configurations are possible, however, and may be accomplished by extending the instructions provided here.

Note: When configuring multiple seats, the best method is to first attach all devices (keyboards, pointing devices, and monitors) and configure the system with a single X server. Once this is done, the configuration may be modified to assign individual subsets of these devices to be managed by separate X servers.

Configuring a system for multi-seat operation involves the following steps, each described in a separate subsection below:

1. Identify the correct event devices (that is, keyboards and pointing devices) for each seat.
2. Create and edit an `XF86Config-`*N* server file for the desired configuration.
3. Point X to the newly-created `XF86Config-`*N* server file.

Identifying Event Devices

An "event device" is a keyboard or pointing device. All event devices connected to the system are listed at boot time on lines beginning with the string "input." These boot messages may be displayed at a Linux command prompt using the `dmesg` command. The output from the `dmesg` command can be quite long, and therefore is usually filtered with a `grep` command. For example:

```
# dmesg | grep ^input
input0: USB HID v1.10 Keyboard [NOVATEK Generic USB Keyboard] on usb1:4.0
input1: USB HID v1.00 Mouse [Logitech N43] on usb1:5.0
input2: USB HID v1.00 Mouse [Logitech N43] on usb1:6.0
input3: USB HID v1.10 Keyboard [NOVATEK Generic USB Keyboard] on usb1:7.0
input4: USB HID v1.00 Keyboard [SILITEK USB Keyboard and Mouse] on usb1:9.0
input5: USB HID v1.00 Mouse [SILITEK USB Keyboard and Mouse] on usb1:9.1
input6: USB HID v1.00 Mouse [Logitech N43] on usb1:10.0
```

All input devices detected during boot-up will have device nodes created for them in the `/dev/input` directory as follows:

- Each keyboard will have an associated event* device node.
- Each pointing device will have both an associated event* device node and an associated mouse* device node.

The mapping of devices to nodes is by number (that is, input0 maps to event0, input1 maps to event1, and so on). The first input that is a pointing device gets mapped to mouse0, the next input that is a pointing device gets mapped to mouse1, and so on.

The `dmesg` output shown above would therefore create the following mapping:

```
input0: event0
input1: event1, mouse0
input2: event2, mouse1
input3: event3
input4: event4
input5: event5, mouse2
input6: event6, mouse3
```

This mapping can then be used to edit the `XF86Config-`*N* server file, as described in the next subsection, "Creating a Multi-Seat XF86Config File" on page 526.

Creating a Multi-Seat XF86Config File

A multiple-X server configuration requires a customized XF86Config file containing a separate ServerLayout section for each X server you will be running.

Note: The original ServerLayout section (always identified as "Main Layout") is typically left unchanged, allowing the system to easily be reconfigured as a single-X server system.

Creating a New XF86Config File

Start out by creating a new XF86Config file. The easiest way to do this is to simply make a copy of the system's regular XF86Config file, as follows:

```
# cp /etc/X11/XF86Config /etc/X11/XF86Config-Nservers
```

N is the number of servers you will be configuring.

Configuring the Input Devices

Next, configure the input devices as follows:

1. Copy the section beginning:

   ```
   Section "InputDevice"
     Identifier "Keyboard1"
   ```

   and insert a duplicate copy (or copies) below the existing section, until there is one copy for each keyboard (including the original copy in this count).

2. Edit all the keyboard InputDevice sections, in each one changing the driver from "keyboard" to "evdev" and adding an Option line identifying the appropriate event device (in this example, "`/dev/input/event0`"). The resulting InputDevice sections will look something like the following:

   ```
   Section "InputDevice"
     Identifier "Keyboard1"
     Driver "evdev"
     Option "Device" "/dev/input/event0"
     # ...
   EndSection
   ```

   Note: See "Identifying Event Devices" on page 525 for instructions on how to determine the appropriate event device for each section.

   Note: You may assign any number of keyboards to a single X server, but no keyboard may be assigned to more than one X server.

3. Copy the section beginning:

   ```
   Section "InputDevice"
     Identifier "Mouse1"
   ```

   and insert a duplicate copy (or copies) below the existing section, until there is one copy for each pointing device (including the original copy in this count).

4. Edit all the mouse InputDevice sections, changing the Option "Device" line from the default "`/dev/input/mice`" to one identifying the appropriate event device (in this example, "`/dev/input/mouse0`"). The resulting InputDevice sections will look something like the following:

   ```
   Section "InputDevice"
     Identifier "Mouse1"
   ```

```
  Driver "mouse"
  Option "Device" "/dev/input/mouse0"
  # ...
EndSection
```

Note: See “Identifying Event Devices” on page 525 for instructions on how to determine the appropriate event device.

Note: You may assign any number of pointing devices to a single X server, but no pointing device may be assigned to more than one X server.

Configuring the New ServerLayout Sections

In this new `XF86Config-Nservers` file, perform the following steps:

1. Copy the section beginning:

   ```
   Section “ServerLayout”
       Identifier “Main Layout”
   ```

   and insert a duplicate copy (or copies) below the existing section, until there is one copy for each X server you will have (do NOT include the original “Main Layout” copy in this count).

2. While leaving the original ServerLayout section identified as “MainLayout,” give each new ServerLayout section a new name. For example, the first server might be named “Layout0”:

   ```
   Identifier “Layout0”
   ```

   the next “Layout1,” and so on.

3. Within each new Server Layout section, disable (delete or comment out) every screen that will not be used in that layout:

```
    Screen "Screen SG-0"        ""      ""      ""      "Screen SG-1"
#    Screen "Screen SG-1"        ""      ""      "Screen SG-0"   ""
```

> **Note:** You may assign any number of screens to a single X server, but no screen may be assigned to more than one X server.

4. Edit each Server Layout section to make sure than no remaining uncommented screen lists as adjacent another screen that will be managed by a different X server:

```
    Screen "Screen SG-0"        ""      ""      ""      ""
#    Screen "Screen SG-1"        ""      ""      "Screen SG-0"   ""
```

5. Within each Server Layout section, change the input devices to the correct ones for that X server. For example, the first X server might use:

   ```
   InputDevice “Mouse1” “CorePointer”
   InputDevice “Keyboard1” “CoreKeyboard”
   ```

6. Save the `XF86Config-Nservers` file.

For an example ServerLayout section from an `XF86Config-3server` file, see “Example “ServerLayout” Sections for Three X Servers” on page 530. In this example, the

first two X servers manage one screen each, while the third X server manages two screens.

Example “ServerLayout” Sections for Three X Servers

```
# **********************************************************************
# ServerLayout sections.
# **********************************************************************

Section "ServerLayout"
    Identifier  "Main Layout"
    Screen "Screen SG-0"        ""      ""      ""      "Screen SG-1"
    Screen "Screen SG-1"        ""      ""      "Screen SG-0"   "Screen SG-2"
    Screen "Screen SG-2"        ""      ""      "Screen SG-1"   "Screen SG-3"
    Screen "Screen SG-3"        ""      ""      "Screen SG-2"   ""
    InputDevice "Mouse1" "CorePointer"
    InputDevice "Keyboard1" "CoreKeyboard"
EndSection

Section "ServerLayout"
    Identifier  "Layout0"
    Screen "Screen SG-0"        ""      ""      ""      ""
    InputDevice "Mouse1" "CorePointer"
    InputDevice "Keyboard1" "CoreKeyboard"
EndSection

Section "ServerLayout"
    Identifier  "Layout1"
    Screen "Screen SG-1"        ""      ""      ""      ""
    InputDevice "Mouse2" "CorePointer"
    InputDevice "Keyboard2" "CoreKeyboard"
EndSection

Section "ServerLayout"
    Identifier  "Layout2"
    Screen "Screen SG-2"        ""      ""      ""      "Screen SG-3"
    Screen "Screen SG-3"        ""      ""      "Screen SG-2"      ""
    InputDevice "Mouse3" "CorePointer"
    InputDevice "Keyboard3" "CoreKeyboard"
EndSection
```

Pointing X to the New XF86Config-*N*server File

Once you have created the new `XF86Config-Nserver` file, the last step is to tell X to use the new layouts it contains, rather than the default server layout. To do so, perform the following steps:

1. Make a backup copy of the default single server `/etc/X11/xdm/gdm.conf` file:

   ```
   # cp /etc/X11/gdm/gdm.conf /etc/X11/gdm/gdm.conf-old
   ```

2. Edit the `/etc/X11/gdm/gdm.conf` file to use the new server layouts you defined in the XF86Config file:

   In the `[servers]` section, comment out the standard server, then add the new server layouts you will be using:

   ```
   #0=Standard
   0=Layout0
   1=Layout1
   2=Layout2
   ```

3. Define each new server layout. For example:

```
[server-Layout0]
name=Layout0 server
command=/usr/X11R6/bin/X :0 -xf86config /etc/X11/XF86Config.3server -layout Layout0
flexible=true
```

   For an example of a multi-X-server `[servers]` section, see "Example /etc/X11/xdm/gdm.conf Servers Section for Three X Servers" on page 532.

4. Save the file and reboot the system.

Example /etc/X11/xdm/gdm.conf Servers Section for Three X Servers

```
[servers]

#0=Standard
0=Layout0
1=Layout1
2=Layout2

[server-Standard]
name=Standard server
command=/usr/X11R6/bin/X
flexible=true

[server-Layout0]
name=Layout0 server
command=/usr/X11R6/bin/X :0 -xf86config /etc/X11/XF86Config.3server -layout Layout0
flexible=true

[server-Layout1]
name=Layout1 server
command=/usr/X11R6/bin/X :1 -xf86config /etc/X11/XF86Config.3server -layout Layout1
flexible=true

[server-Layout2]
name=Layout2 server
command=/usr/X11R6/bin/X :2 -xf86config /etc/X11/XF86Config.3server -layout Layout2
flexible=true
```

Index

B

C

D

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

W

X

Z