OpenGL® Porting Guide

Document Number 007-1797-030

CONTRIBUTORS

Written by C J Silverio, Beth Fryer, and Jed Hartman. Revised by Renate Kempf.
Edited by Christina Cary
Production by Mike Dixon
Engineering contributions by Kurt Akeley, Allen Akin, Gavin Bell, Derrick Burns, Dave Ciemiewicz, Tom Davis, Chris Frazier, Paul Ho, Phil Karlton, Reuel Nash, Mark Segal, Dave Shreiner, Rolf van Widenfelt, and Mason Woo. Revised example programs by Paul Ho.

OpenGL® Porting Guide
Document Number 007-1797-030

Contents

List of Figures

List of Tables

About This Guide

This guide tells you how to port your existing IRIS GL code to OpenGL. It

- describes how to use the *toogl* automatic translation script
- lists OpenGL equivalents for IRIS GL calls
- describes how to reimplement IRIS GL windowing code using the X Window System and IRIS IM APIs (IRIS IM is the Silicon Graphics port of the industry-standard OSF/Motif software)
- provides basic information for working with the X Window System

This guide is for developers who have been using IRIS GL. It is not an introduction to graphics programming and it is not comprehensive OpenGL documentation. For more complete OpenGL documentation, see "OpenGL Documentation" on page xv.

Note: This guide is written for programmers who are working in C. It doesn't discuss OpenGL Fortran and Ada wrappers.

What This Guide Contains

This guide includes the following chapters:

- Chapter 1, "Introduction to Porting From IRIS GL to OpenGL," describes some of the major differences between IRIS GL and OpenGL, lists the Silicon Graphics tools you can use to facilitate the transition, and provides some general porting instructions.
- Chapter 2, "Using the toogl Tool," explains how to use the automatic translation tool, which can do much of the porting work for you.
- Chapter 3, "After toogl: How to Finish Porting to OpenGL," discusses IRIS GL commands that might need some extra porting attention, giving command equivalents and providing porting tips for each.

- Chapter 4, “OpenGL in the X Window System,” describes two methods for using the X Window System™ to manage windows and events with OpenGL: using Xt and the Silicon Graphics IRIS IM widget or using Xlib.
- Appendix A, “OpenGL Commands and Their IRIS GL Equivalents,” is a complete alphabetical list of IRIS GL calls and their OpenGL equivalents (if an equivalent exists) along with cross-references to documentation, where available.
- Appendix B, “Differences Between OpenGL and IRIS GL,” provides a more complete list of the differences between OpenGL and IRIS GL than Chapter 1 offers.
- Appendix C, “OpenGL Names, Types, and Error,” explains OpenGL naming conventions, lists OpenGL defined types, and describes error handling in OpenGL.
- Appendix D, “Example OpenGL Program With the GLUT Library,” provides an example OpenGL program that uses the GLUT library for windowing and event handling.
- Appendix E, “Example Program Using Xt and a WorkProc,” provides an example OpenGL program using Xt, IRIS IM, and the Silicon Graphics widget. The program demonstrates the use of a WorkProc for animation.
- Appendix F, “Example Mixed-Model Programs With Xlib,” provides two example mixed-model programs using Xlib. Each program is shown in both IRIS GL and OpenGL form.

Where to Get More Information

As you use this guide, you will probably have to refer to the OpenGL reference pages, the IRIS GL reference pages, and the programming guides. You can read all the reference pages online using the *man* command, or you can buy the printed OpenGL reference pages. These are published in the *OpenGL Reference Manual*, available in bookstores (see “OpenGL Documentation” on page xv).

Note: If you’re viewing this manual online using IRIS InSight, click any red underlined reference page name to view the reference page.

In addition, you may find the OpenGL documentation, GLUT documentation, IRIS GL documentation, and window system documentation that's listed in the following sections helpful:

- "OpenGL Documentation" on page xv
- "GLX and GLUT Documentation" on page xvi
- "X Window System Documentation" on page xvi
- "OSF/Motif Documentation" on page xvii

OpenGL Documentation

For more information on programming in OpenGL 1.1, refer to these manuals:

- OpenGL Architecture Review Board; Renate Kempf and Chris Frazier, editors. *OpenGL Reference Manual*. *The Official Reference Document for OpenGL, Version 1.1*. Reading, MA: Addison Wesley Longman Inc. 1996. ISBN 0-201-46140-4
- Woo, Mason, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*: *The Official Guide to Learning OpenGL, Version 1.1*. Reading, MA: Addison Wesley Longman Inc. 1997. ISBN 0-201-46138-2
- *OpenGL on Silicon Graphics Systems* (Silicon Graphics manual; document number 007-2392-002)

For more information on programming in OpenGL 1.0, refer to these manuals:

- OpenGL Architecture Review Board. *OpenGL Reference Manual*. Reading, MA: Addison-Wesley Publishing Company. 1992. ISBN 0-201-63276-4
- Neider, Jackie, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Reading, MA: Addison-Wesley Publishing Company. ISBN 0-201-63274-8
- *OpenGL on Silicon Graphics Systems* (Silicon Graphics manual; document number 007-2392-001)

GLX and GLUT Documentation

- Kilgard, Mark J. *OpenGL Programming for the X Window System*. Menlo Park, CA: Addison-Wesley Developer's Press. 1996. ISBN 0-201-48369-9
- Pointers to the GLUT Library, to interesting technical papers, and to the comp.graphics.opengl mailing list are on the OpenGL home page: http://www.openGL.org/

IRIS GL Documentation

For more information on programming with IRIS GL, refer to these Silicon Graphics manuals:

- *Graphics Library Programming Guide,* Volume 1 (document number 007-1210-060)
- *Graphics Library Programming Guide,* Volume 2 (document number 007-1702-020)
- *Graphics Library Programming Tools and Techniques* (document number 007-1489-030)

X Window System Documentation

For comprehensive information on the X Window System, Xlib, Xt, and X protocol, see the Digital Press X Series:

- Scheifler, Robert W., and James Gettys, et al. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*. Third Edition, X Version 11, Release 5, Burlington, MA. Digital Press—Digital Equipment Corporation, 1992. ISBN 1-55558-088-2
- Asente, Paul J., and Ralph R. Swick. *X Window System Toolkit: The Complete Programmer's Guide and Specification*. Burlington MA: Digital Press—Digital Equipment Corporation. 1992. ISBN 1-55558-051-3

Or refer to the O'Reilly X Window System Series, Volumes 1, 4, and 5:

- Nye, Adrian. Volume One: *Xlib Programming Manual*. Sebastopol, CA. O'Reilly & Associates. 3rd edition July 1992. ISBN 1-56592-002-3
- Nye, Adrian, and Tim O'Reilly. Volume Four: *X Toolkit Intrinsics Programming Manual*. Sebastopol, CA: O'Reilly & Associates. Second edition, 1992. ISBN 1-56592-013-9

- Flanagen, David (editor). Volume Five: *X Toolkit Intrinsics Reference Manual*. Sebastopol, CA: O'Reilly & Associates. Third edition, April 1992. ISBN 1-56592-007-4

OSF/Motif Documentation

For information on OSF/Motif, see the Prentice-Hall OSF/Motif series:

- *OSF/Motif Programmer's Guide*, Open Software Foundation, PTR Prentice-Hall, Inc., Englewood Cliffs, NJ.
- *OSF/Motif Programmer's Reference*, Open Software Foundation, PTR Prentice-Hall, Inc., Englewood Cliffs, NJ.
- *OSF/Motif Style Guid*e, Open Software Foundation, PTR Prentice-Hall, Inc., Englewood Cliffs, NJ.

Conventions Used in This Guide

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

Typographical Conventions

This guide uses the following typographical conventions:

Italics	Filenames, IRIX command names, function parameters, and book titles.
`Fixed-width`	Code examples and system output.
Bold	Function names, with parentheses following the name—for example **glPolygonMode()**—and arguments to command-line options.

Note: In all tables, regular font is used for function names. This avoids clutter in the table, which would make it difficult to read.

Function Naming Conventions

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

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

GLX*()	IRIS GL mixed-model support
Glx*()	IRIS GL support for IRIS IM
glX*()	OpenGL support for X
GLw*()	OpenGL support for IRIS IM

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

```
IRIS% man glxgetconfig
```

displays both reference pages, one following the other.

Changes in This Version of the Document

This version of the document has been updated in the following ways:

- All references to the aux library have been removed and replaced with references to the GLUT library where appropriate. The aux library was never intended for production-level code. Its removal from this document intends to discourage programmers from continuing to use it.
- All examples were updated to use the glut library instead of the aux library.

- This document is targeted at both OpenGL 1.0 and OpenGL 1.1 users. It refers to OpenGL 1.1 functions where new functionality has been introduced (old references to extensions remain where appropriate).
- Most references to reference pages are now links when viewed with IRIS Insight.
- Chapter and section overviews with links to sections or subsections have been added.
- Appendix B, Differences Between OpenGL and IRIS GL, was updated to contain the most recent information supplied by Mark Kilgard.

Chapter 1

Introduction to Porting From IRIS GL to OpenGL

This chapter provides an overview to porting from IRIS GL to OpenGL. It discusses the following topics:

- "Differences Between IRIS GL and OpenGL" discusses window and event management, some fundamental implementation differences, and naming conventions.
- "Tools and Libraries to Help Port Your Code" on page 3 briefly explores the toogl tool, Motif drawing area widgets, and the glx library.
- "Porting IRIS GL Programs to OpenGL" on page 4 explains how to port both a mixed-model program (using IRIS GL and X) and an IRIS GL program that doesn't use X.
- "If You're Not Porting Your Code to OpenGL Yet" on page 6 provides some advice for programmers who don't plan to port to OpenGL immediately.

Differences Between IRIS GL and OpenGL

One focus of OpenGL is portability. OpenGL and IRIS GL therefore differ in several major areas. This section lists a few important ways in which OpenGL is different from IRIS GL. A more complete list of the differences between the two languages is provided in Appendix B, "Differences Between OpenGL and IRIS GL."

Here are some key differences between OpenGL and IRIS GL:

- **Window Management**. OpenGL is window-system independent. It therefore contains no windowing, pop up menus, event handling, color-map loading, buffer allocation and management, font file formats, or cursor handling. These functions are delegated to the window or operating system. You can use the GLUT library for simple window handling under X. If you need more sophisticated windowing and event handling calls, you have to include the relevant X Window System calls in your program. Silicon Graphics provides some special GLX calls, where OpenGL rendering is made available as an extension to X in the formal X sense, and an

widget to help you replace your IRIS GL windowing, event, and colormap handling calls. See Chapter 4, "OpenGL in the X Window System," for details.

- **Naming Conventions.** OpenGL establishes and adheres to a standard "name space." OpenGL commands begin with the **gl** prefix (**glEnable()**, **glTranslatef()**, and so on). This convention prevents conflict with commands from other libraries. "OpenGL Command Names" on page 153 explains the OpenGL naming conventions, and "OpenGL Defined Types" on page 155 lists the OpenGL defined types with their C data type equivalents.
- **State Variables**. Like IRIS GL, OpenGL maintains state variables for color, fog, texture, lighting, viewport, and so on. But OpenGL manages state variables more directly and consistently than IRIS GL does. With OpenGL there are no tables—you just load values directly.

  Because OpenGL doesn't keep tables of predefined lights and materials, it has no equivalent for "binds," although you can use display lists to get a similar effect. "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67 explores different approaches to replacing display lists. Refer also to "Porting Lighting and Materials Calls" on page 68 and "Porting Texture Calls" on page 73 for more discussion and some examples.
- **Display Lists**. OpenGL display lists are not editable. In OpenGL, the sole purpose of display lists is to efficiently cache OpenGL commands. As a result, IRIS GL calls for editing display lists have no OpenGL equivalent. If your IRIS GL program edits display lists, you have to reimplement it to some extent. "Porting Display Lists" on page 63 lists the relevant IRIS GL calls, and "Achieving Edited Display List Behavior" on page 65 provides some suggestions for porting code that edits display lists.
- **Fonts**. IRIS GL provides calls to handle fonts and text strings. Although OpenGL can render text, it doesn't provide a file format for fonts. For fonts and text strings, you can use the GLX call **glXUseXFont()** in conjunction with the OpenGL calls **glCallLists()** and **glListBase()**. "Fonts and Strings" on page 92 provides suggestions for porting fonts and strings.
- **Utility Library.** OpenGL provides a utility library, called the GL Utility Library (GLU), that contains additional routines (such as NURBS and quadric surfaces rendering routines). This library is discussed in the *OpenGL Programming Guide*. Reference pages for all the routines the GLU consists of are included in the *OpenGL Reference Manual*. These routines all begin with the "glu" prefix (**gluDisk()**, **gluErrorString()**, and so on).

Tools and Libraries to Help Port Your Code

Silicon Graphics provides tools and libraries to help you port your IRIS GL program:

- The ***toogl* tool** translates your program's IRIS GL calls to OpenGL calls. *toogl* does do a lot of the translation work for you but it can't translate everything (in particular, it can't translate windowing and event calls). You therefore have to edit the output. Chapter 2, "Using the toogl Tool," explains how to use **toogl**.
- The **OpenGL extension to X (GLX)** provides a variety of routines to help you replace your old IRIS GL windowing, event, and font calls. Chapter 4 explains how to use GLX. Reference pages for the GLX routines are included in the *OpenGL Reference Manual*. Consider looking at the glXIntro reference page first for an overview.
- The **GLwDrawingArea and GLwMDrawingArea widgets** help you port your code to run in an X window. These widgets provide a window with the appropriate visual and color maps needed for OpenGL, based on supplied parameters. They also provide callbacks for redraw, resize, input, and initialization. For information on how to use these widgets, see Chapter 4.
- The **GLUT Library** is a programming interface with ANSI C and FORTRAN bindings for writing window system independent OpenGL programs. The toolkit supports the following functionality:
  - Multiple windows for OpenGL rendering
  - Callback-driven event processing
  - Sophisticated input devices
  - An "idle" routine and timers.
  - A simple cascading pop-up menu facility
  - Utility routines to generate various solid and wire frame objects
  - Support for bitmap and stroke fonts
  - Miscellaneous window management functions, including managing overlays

  You can find the GLUT library (and the associated documentation) on the OpenGL home page http://www.opengl.org, or the Silicon Graphics OpenGL page, http://www.sgi.com/Technology/openGL/. It is also discussed in *OpenGL Programming for the X Window System*; see "Where to Get More Information" on page xiv.

Porting IRIS GL Programs to OpenGL

This section lists three porting scenarios. Select the one that best matches your situation and complete the porting tasks listed. More information is provided in subsequent chapters. This overview discusses the major porting tasks for different kinds of programs:

- "Porting IRIS GL Programs That Use X Calls" on page 4
- "Porting IRIS GL Programs With Simple Windowing" on page 5
- "Porting IRIS GL Programs With Complex Windowing" on page 5

In all cases, after you've finished the porting tasks listed, you probably have to iteratively compile, run, and debug your program. If necessary, run the *toogl* script again to catch any IRIS GL commands that you missed. You may find it useful to refer to "Error Handling" on page 156 which gives some basic information on error handling in OpenGL.

Porting IRIS GL Programs That Use X Calls

If your IRIS GL program uses X for all window system calls, including windowing and event handling, it will be relatively easy to port it to OpenGL. Here's what you have to do:

1. Run your program through a C beautifier (such as *cb*).
2. Run the *toogl* filter script on your code.
3. Edit *toogl* output. See Chapter 2 for a list of known trouble spots where you have to port your code explicitly. See Chapter 3, "After toogl: How to Finish Porting to OpenGL," for specific suggestions.
4. Convert your IRIS GL X Window System calls to GLX calls.
   - If you used one of the Motif widgets, GlxDraw or GlxMDraw, switch to the OpenGL version: GLwDrawingArea or GLwMDrawingArea. Chapter 4 discusses mixed-model programming in OpenGL and provides information about the OpenGL version of the Silicon Graphics mixed-model widget.
   - If you didn't use a widget, look at Appendix F, "Example Mixed-Model Programs With Xlib."

- The *OpenGL Reference Manual* contains an overview of the OpenGL Extension to the X Window System. It also includes a glXIntro reference page and reference pages for all the OpenGL/X routines.
- *OpenGL Programming for the X Window System* by Mark Kilgard discusses all aspects of using OpenGL in the X Window System environment. See "Where to Get More Information" on page xiv for complete bibliographical information.

Porting IRIS GL Programs With Simple Windowing

If your program doesn't use X Window System calls but does use simple windowing, you can probably use the GLUT library to replace IRIS GL windowing, color map, and event handling calls. This is possible if your code meets the following conditions:

- Is reasonably simple
- Conforms to Silicon Graphics recommendations
- Doesn't use unsupported calls,

Here's what you have to do:

1. Replace windowing and event handling calls with GLUT calls.
2. Run your program through a C beautifier (such as *cb*).
3. Run the *toogl* filter script on your code.
4. Edit toogl output. See Chapter 2 for a list of known trouble spots. You will probably have port some of the trickier commands explicitly; see Chapter 3 for specific suggestions.

Porting IRIS GL Programs With Complex Windowing

If your IRIS GL code doesn't use X windowing calls and your windowing and event handling code uses unsupported calls, doesn't conform to Silicon Graphics recommendations, or is complicated or unusual in scope, using the X Window System is the best solution.

Here's what you have to do:

1. Run your program through a C beautifier (such as *cb*),
2. Run the *toogl* filter script on your code.

3. Edit *toogl* output. See Chapter 2 for a list of known trouble spots. You will probably have port some of the trickier commands explicitly; see Chapter 3 for specific suggestions.
4. Port your program to use OpenGL and X. You can do this either by using Xlib and directly replacing calls like **winopen()** and **qread()** with their GLX equivalents, or by using Xt along with a widget set and the OpenGL widget GLwDrawingArea. See Chapter 4 for more information.

If You're Not Porting Your Code to OpenGL Yet

If you're not porting to OpenGL now, but know that you will be porting in the future, it's a good idea to switch to mixed-model mode now.

- Replace all GL windowing calls with GLX and X calls.
- Replace GL event handling with X event handling. Refer to the *Graphics Library Programming Tools and Techniques* manual for detailed instructions.
- Learn what IRIS GL features have no OpenGL equivalents. Avoid using them in new code, and reimplement code that does use them. Appendix A, "OpenGL Commands and Their IRIS GL Equivalents," lists IRIS GL commands and indicates which commands are not supported in OpenGL.
- Replace any obsolete or unsupported calls with newer IRIS GL equivalents as soon as possible.

Chapter 2

Using the toogl Tool

toogl (which stands for To OpenGL and is pronounced TOO-guhl) is a script that takes IRIS GL code as input and produces commented, nearly equivalent OpenGL code as output.

This chapter explores how to use and get the most from *toogl*. It explains where to find a copy of *toogl* and how to use *toogl* most effectively. It also mentions some areas of your IRIS GL code that might give you problems.

The chapter discusses the following topics:

- "Getting Started with toogl" explains how you can get a copy of *toogl* and then explores calling *toogl* for one file and in batch mode.
- "Using xdiff or gdiff to Compare Files" on page 9 explains two tools that are helpful for comparing your original IRIS GL file with the OpenGL output produced by *toogl*.
- "Using toogl Effectively" on page 10
- "Editing toogl Output: Areas that Need Special Attention" on page 10
- "Editing toogl Output: An Example" on page 15

Getting Started with toogl

You can use *toogl* to do much of the work of translating your IRIS GL code to OpenGL code. While *toogl* can't do everything, it can do all the tedious work of changing command names, and it can call your attention to code you have to port explicitly.

This section first explains "Finding and Building toogl," then provides the syntax in "Calling toogl," and explores "Using toogl in Batch Mode." The last section briefly discusses "What toogl Will and Won't Do for You."

Finding and Building toogl

A copy of *toogl* is in the */usr/share* directory. The exact location depends on the operating system version you are using. If you want to look at the source (it's in C++), you can get a copy from the OpenGL directory.

You can also get a copy of the tool from the "Contributed Tools" area of the Silicon Graphics OpenGL home page (http://www.sgi.com/Technology/openGL/)

To build the tool, enter:

```
# setenv OBJECT_STYLE 32
# make
```

Warning: If you build toogl with OBJECT_STYLE set to n32 (the default on current systems), the tool does not work!

Calling toogl

toogl syntax:

```
toogl [-cwq] < infile > outfile
```

You can use any of these options with *toogl*:

-c	Don't clutter up the output with comments
-w	Don't remove window manager calls like **winopen()**, **mapcolor()**
-q	Don't remove event queue calls like **qread()**, **setvaluator()**

Note: *toogl* doesn't attempt to translate event queue and windowing calls. It simply removes them, and replaces them with warning comments. The **-w** and **-q** flags merely suppress the comments.

Keep your original source! Accidents happen.

Using toogl in Batch Mode

To process a directory full of source files automatically, you could use a shell script like the following:

```
#!/bin/sh
mkdir OpenGL
for i in *.c
   do
      echo "Converting " $i " ..."
      toogl < $i > OpenGL/$i
   done
```

What toogl Will and Won't Do for You

toogl is a filter that scans each line of an input file, looking for IRIS GL calls. When *toogl* finds an IRIS GL function, it replaces the function with the corresponding OpenGL function(s).

Because *toogl* can't translate everything, you have to edit its output. Any time *toogl* translates code that you may have to look at, check, or change, it marks the potential problem with a comment starting with "OGLXXX". (You can use the **-c** option to suppress the comments.)

Using xdiff or gdiff to Compare Files

After you've converted your program using *toogl*, you have to edit the *toogl* output. You'll probably want to examine the differences between the *toogl* output and your original program—or any other version of the program. To do so, use either *gdiff* or *xdiff*.

To use *gdiff*, enter:

`gdiff -b` *file1 file2*

where *file1* and *file2* are the names of the files you want to compare. The **-b** option tells *gdiff* to ignore trailing blanks on lines when comparing files. You might also want to use the **-w** option, which tells *gdiff* to ignore white space.

To use *xdiff*, enter:

`xdiff` *file1 file2*

See the xdiff reference page for more information.

Using toogl Effectively

Here are a few suggestions for getting the most out of *toogl*:

- For best results, use a C beautifier (such as *cb*) on your code before running *toogl*.
- Use *gdiff* (or *xdiff*) to browse through the source and the translation simultaneously.
- *toogl* expects to find the matching parentheses or quotes on the same line as the IRIS GL function.
- *toogl* expects to find only spaces and tabs between a function name and the opening parenthesis. For example, the code

  ```
  v3f
  (foo);
  ```

  will be left unchanged, as will

  ```
  v3f /* comment */ (foo);
  ```

  Running a C beautifier on your program before using *toogl* can prevent problems like this.
- *toogl* expects that C comments inside the argument list of a function don't contain parentheses or quote characters. Faced with the following code, *toogl* will generate a warning and will not do a translation:

  ```
  v3f ( foo /* I really mean bar “-) */ );
  ```

Editing toogl Output: Areas that Need Special Attention

After you've run *toogl* on your code, you have to edit the output. Some areas are more problematic than others—for example, **v()** calls usually translate quite neatly into **glVertex()** calls, but texture calls often don't translate well. This section lists some of the general areas that are likely to need special attention. Chapter 3, "After toogl: How to Finish Porting to OpenGL," provides more detailed information on problem areas.

This section briefly addresses the following issues:

- "Windowing, Device, and Event Calls"
- "Parentheses and Quotes"
- "Defined Color Constants"
- "clear() Calls"
- "Get Calls"
- "rotate() Calls"
- "swaptmesh() Calls"
- "Texturing Calls"
- "def/bind Calls"
- "Calls Without Direct Equivalents"
- "Finding OpenGL Replacements for IRIS GL Calls"
- "Performance"

Windowing, Device, and Event Calls

toogl can't translate sections of code that make window manager, window configuration, device, or event calls, or that load a color map. You have to rewrite these code sections yourself. You can use the **-w** and **-q** options to make *toogl* leave this code untouched, so you can still read it to translate it manually. If your windowing and event handling calls are simple and straightforward, you can replace them with calls from the GLUT library. For more information on GLUT, see the OpenGL home page http://www.OpenGL.org or the Silicon Graphics OpenGL page http://www.sgi.com/Technology/openGL/. See also "GLX and GLUT Documentation" on page xvi.

If your windowing and event handling calls are fairly sophisticated, you have to incorporate the OpenGL code into X Windows directly, either using the OpenGL widget or using Xlib. This is explained in Chapter 4, "OpenGL in the X Window System."

Parentheses and Quotes

toogl understands a little about matching parentheses and quotes. It translates

```
v3f( v[strlen(strcat(foo, "foo("))] );
```

into

```
glVertex3fv( v[strlen(strcat(foo, "foo("))] );
```

Defined Color Constants

IRIS GL provides defined color constants: BLACK, BLUE, RED, GREEN, MAGENTA, CYAN, YELLOW, and WHITE. OpenGL does not provide these constants and *toogl* does not translate them, so you have to port them explicitly.

clear() Calls

Make sure **clear()** calls are correctly translated. For example, assume your program clears the window as follows:

```
color(BLACK);
clear();
```

toogl translates those two lines as follows:

```
glIndex(BLACK);
glClear(GL_COLOR_BUFFER_BIT);
```

The above code fragment is incorrect for these reasons:

- OpenGL does not provide the color constant BLACK
- OpenGL maintains a clear color that's distinct from the drawing color.

A better translation is the following:

```
glIndex(0);
glClearIndex(0);
glClear(GL_COLOR_BUFFER_BIT);
```

Get Calls

toogl does not always translate IRIS GL "get" calls (such as **getdepth()**, **getcolor()**, and so on) correctly. *toogl* translates

```
i = getcolor();
getdepth(&near, &far);
```

into

```
/* OGLXXX replace value with your variable */
i = glGetIntegerv(GL_CURRENT_INDEX, &value);

/* OGLXXX You can probably do better than this. */
{
  int get_depth_tmp[2];
  glGetIntegerv(GL_DEPTH_RANGE, get_depth_tmp);
  *(&near)=get_depth_tmp[0];
  *( &far)=get_depth_tmp[1];
};
```

This guide lists the **glGet*()** calls related to a particular topic in the section on that topic. For general information on replacing get calls, see "Porting IRIS GL get* Commands" on page 20.

rotate() Calls

The OpenGL rotation call **glRotate()** is different from **rotate()**. You will probably have to modify the code after translating the program with *toogl*. See "Porting Matrix and Transformation Calls" on page 24 for details.

swaptmesh() Calls

OpenGL has no equivalent for **swaptmesh()**; *toogl* flags occurrences of the function and leaves it up to you to restructure your triangles.

Texturing Calls

toogl correctly translates texture coordinate calls. You have to do additional work, as explained in "Porting Texture Calls" on page 73.

Note: If you're using OpenGL 1.1, you can take advantage of subtextures. Because *toogl* was developed for OpenGL 1.0, you have to implement the use of OpenGL subtextures yourself.

def/bind Calls

OpenGL does not keep tables of predefined lights and materials, so it has no equivalent for "binds." You can use display lists to mimic the behavior. See "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67 for more information. See also "Porting Lighting and Materials Calls" on page 68 and "Porting Texture Calls" on page 73 for more discussion and some examples.

Calls Without Direct Equivalents

There are some IRIS GL calls that *toogl* cannot directly translate into OpenGL calls. **arcf()** is one example. You have to port such calls explicitly. "Editing toogl Output: An Example" on page 15 gives an example for porting a call like **arcf()**.

Finding OpenGL Replacements for IRIS GL Calls

Appendix A, "OpenGL Commands and Their IRIS GL Equivalents," contains a table listing IRIS GL commands and the corresponding OpenGL commands, and tells you where to go for more information. This table also indicates which IRIS GL calls are not supported in OpenGL.

Performance

toogl doesn't necessarily produce fast OpenGL code; in fact, such an automatic port usually results in loss of performance. Details of improving OpenGL performance are beyond the scope of the current edition of this guide; however, you can find some specific tips in "Porting Screen and Buffer Clearing Commands" on page 23 and "Porting Lighting and Materials Calls" on page 68.

Note: The performance chapters of *OpenGL on Silicon Graphics Systems* provide additional information.

Two features of OpenGL that can drastically improve performance are display lists and direct rendering. Use these features whenever possible in OpenGL programs. For information on display lists, see "Porting Display Lists" in Chapter 3. For information on direct rendering, see the glXCreateContext reference page. If you aren't careful, you may set up indirect rendering without noticing that it's indirect; specify direct rendering explicitly where possible.

Here are few more tips:

- If you're drawing independent triangles, there's no need to put **glBegin()** and **glEnd()** around each set of three vertices. Instead, call `glBegin(GL_TRIANGLES)` and then list as many individual triangles as you need before the **glEnd()**. This optimization alone can noticeably improve performance.
- If you aren't using the depth buffer, disable it. This is particularly important when you call **glDrawPixels()** or other non-3D drawing functions.
- Disable texturing when you call **glDrawPixels()** or any other function that shouldn't use textures. Otherwise, the texture overhead slows down drawing even if you're only drawing a bitmap.

Editing toogl Output: An Example

This section provides an example for working with *toogl* output.

toogl translates the call

```
arcf(1.0, 1.0, 0.9, 1200, 2200);
```

as

```
/* OGLXXX see gluPartialDisk man page */
gluPartialDisk( *gobj, innerRad, outerRad, slices, loops, startAng, endAng);
```

The IRIS GL call **arcf()** can't be directly translated into an OpenGL call. The GL Utility Library call **gluPartialDisk()** is the nearest equivalent, but you have to fill in its arguments explicitly. Compare the reference pages for the two commands, or refer to the section in this guide that discusses porting that command (in this case, "Porting Arcs and Circles" on page 42).

Those materials explain that you have to account for the following changes:

- Arcs are now quadrics and are drawn using quadric objects.
- Angles are now measured in degrees instead of tenths of degrees.
- Instead of specifying a center for your arc in the call, you now do a translation first.
- Angles are now measured on different coordinate axes. The second angle is now a sweep angle instead of an end angle.

Your completed **arcf()** translation might look like this:

```
gluQuadricObj *arcObj;

arcObj = gluNewQuadric(void);
glTranslatef( 1.0, 1.0, 0.0 );
gluPartialDisk( *arcObj, 0.0, 0.9, 100, 2, -30, -100);
```

Chapter 3

After *toogl*: How to Finish Porting to OpenGL

After you run your IRIS GL program through *toogl*, you can use this chapter to find out how to replace IRIS GL calls that *toogl* didn't translate completely or correctly. To get the most out of this discussion, refer to the reference pages as necessary.

The chapter discusses these topics:

Header Files

toogl doesn't replace header files for you, so you have to replace them yourself. This section lists the files your IRIS GL program probably used and the OpenGL files to replace them with.

Table 3-1 Include Lines in IRIS GL and OpenGL Programs

IRIS GL Include Lines	OpenGL Include Lines
#include <gl/gl.h>	#include <GL/gl.h>
#include <gl/device.h>	#include <GL/glu.h>
#include <gl/get.h>	/* X header files start here*/
	#include <Xm/Xm.h>
	#include <Xm/Frame.h>
	#include <Xm/Form.h>
	#include <X11/StringDefs.h>
	#include <X11/keysym.h>

If you use the GLUT library, you also have to include the GLUT header file:

```
#include <GL/glut.h>
```

If you use the Motif widget, replace the include lines as indicated in Table 3-2.

Table 3-2 Include Lines for IRIS GL and OpenGL Motif Widgets

IRIS GL Motif Widget Include Lines	OpenGL Motif Widget Include Lines
For the IRIS IM version of the widget: `#include <X11/Xirisw/GlxMDraw.h>`	For the IRIS IM version of the widget: `#include <GL/GLwMDrawA.h>`
For the generic version of the widget: `#include <X11/Xirisw/GLxDraw.h>`	For the generic version of the widget: `#include <GL/GLwDrawA.h>`

If you're using Xlib and OpenGL/X calls, add

```
#include <GL/glx.h>
```

Porting greset()

OpenGL replaces the functionality of **greset()** with the commands **glPushAttrib()** and **glPopAttrib()**. Use these commands to save and restore groups of state variables.

Call **glPushAttrib()** to indicate which groups of state variables to push onto an attribute stack by taking a bitwise OR of symbolic constants, as follows:

```
void glPushAttrib( GLbitfield mask );
```

The attribute stack has a finite depth of at least 16.

The **glPushAttrib()** and **glPopAttrib()** calls push and pop the server attribute stacks. In OpenGL 1.1, you can also use the functions **glPushClientAttrib()** and **glPopClientAttrib()** to push and pop the client attribute stack.

Each constant refers to a group of state variables. Table 3-3 shows the attribute groups with their corresponding symbolic constant names. For a complete list of the OpenGL state variables associated with each constant, see the glPushAttrib reference page.

Table 3-3 State Attribute Groups

Attribute	Constant
accumulation buffer clear value	GL_ACCUM_BUFFER_BIT
color buffer	GL_COLOR_BUFFER_BIT
current	GL_CURRENT_BIT
depth buffer	GL_DEPTH_BUFFER_BIT
enable	GL_ENABLE_BIT
evaluators	EGL_VAL_BIT
fog	GL_FOG_BIT
GL_LIST_BASE setting	GL_LIST_BIT
hint variables	GL_HINT_BIT
lighting variables	GL_LIGHTING_BIT
line drawing mode	GL_LINE_BIT

Table 3-3 (continued) State Attribute Groups

Attribute	Constant
pixel mode variables	GL_PIXEL_MODE_BIT
point variables	GL_POINT_BIT
polygon	GL_POLYGON_BIT
polygon stipple	GL_POLYGON_STIPPLE_BIT
scissor	GL_SCISSOR_BIT
stencil buffer	GL_STENCIL_BUFFER_BIT
texture	GL_TEXTURE_BIT
transform	GL_TRANSFORM_BIT
viewport	GL_VIEWPORT_BIT
—	GL_ALL_ATTRIB_BITS
pixel storage modes	GL_CLIENT_PIXEL_STORE_BITS (1.1 only)
vertex arrays and enables	GL_CLIENT_VERTEX_ARRAY_BIT (1.1 only)

To restore the values of the state variables to those saved with the last **glPushAttrib()**, call **glPopAttrib()**. The variables you didn't save remain unchanged.

Porting IRIS GL get* Commands

IRIS GL get* calls are of this form:

```
int get thing();
int get things( int *a, int *b);
```

Your IRIS GL program probably includes calls that look like the following:

```
thing = get thing();
if(get thing() == THING) { /* stuff */ }
get things (&a, &b);
```

OpenGL uses **glGet*()** calls for equivalent functionality; they look like this:

```
void glGetIntegerfv(NAME_OF_THING, &thing);
```

Table A-1 on page 109 lists the IRIS GL get functions with their OpenGL equivalents.

In general, this guide lists various parameters for **glGet*()** calls in the sections that discuss topics related to those parameters. To see the parameter values related to matrices, for example, see "Porting Matrix and Transformation Calls" on page 24.

There are other functions to query the OpenGL state, such as **glGetClipPlane()** and **glGetLight()**. These commands are discussed in the sections on related calls, and also in the reference pages.

About glGet*()

There are four types of **glGet*()** functions:

- **glGetBooleanv()**
- **glGetIntegerv()**
- **glGetFloatv()**
- **glGetDoublev()**

The functions have this syntax:

```
glGet<Datatype>v( value, *data )
```

value is of type GL*enum* and *data* of type GL*datatype*. If you issue a **glGet*()** call that returns types different from the expected types, each type is converted appropriately. For a complete list of parameters, see the glGet reference page.

In addition to the basic **glGet*()** function, there are a number of special purpose information retrieval functions: glGetClipPlane, glGetError, glGetLight, glGetMap, glGetMaterial, glGetPixelMap, glGetPointerv, glGetPolygonStipple, glGetString, glGetTexEnv, glGetTexGen, glGetTexImage, glGetTexlevelParameter, glGetTexParameter.

glGet*() Conventions Used in This Book

For the sake of brevity, this guide usually shortens the reference to the form

`glGet*(`*GL_GET_TYPE*`)`

For example,

`glGetIntegerv(GL_VIEWPORT, *`*params*`);`

is abbreviated as

`glGet*(GL_VIEWPORT);`

in tables and text (but not in code examples).

Porting Commands That Required Current Graphics Positions

OpenGL doesn't maintain a current graphics position. IRIS GL commands that depend on the current graphics position, such as **move()**, **draw()**, and **rmv()**, have no equivalents in OpenGL.

Older versions of IRIS GL included drawing commands that relied upon the current graphics position, though their use was discouraged in more recent versions. You have to reimplement parts of your program if you relied on the current graphics position in any way, or used any of the following routines:

- **draw()** and **move()**
- **pmv()**, **pdr()**, and **pclos()**
- **rdr()**, **rmv()**, **rpdr()**, and **rpmv()**
- **getgpos()**

OpenGL has a concept of raster position that corresponds to the IRIS GL current character position. See "Porting Pixel Operations" on page 46 for more information.

Porting Screen and Buffer Clearing Commands

OpenGL replaces a variety of IRIS GL **clear()** calls (such as **zclear()**, **aclear()**, **sclear()**, and so on) with one: **glClear()**. Specify exactly what you want to clear by passing masks to **glClear()**.

When porting screen and buffer clearing commands, consider the following issues:

- OpenGL maintains clear colors separately from drawing colors, with calls like **glClearColor()** and **glClearIndex()**. Be sure to set the clear color for each buffer before making a clear call.
- Since *toogl* has no concept of context, it cannot correctly translate color calls immediately preceding clear calls into **glClearColor()** calls. You have to translate these calls explicitly. For example, suppose your program clears the viewport as follows:

  ```
  color(BLACK);
  clear();
  ```

  toogl translates those two lines as follows:

  ```
  glIndex(BLACK);
  glClear(GL_COLOR_BUFFER_BIT);
  ```

  A better translation of this fragment the following:

  ```
  glClearIndex(0);
  glClear(GL_COLOR_BUFFER_BIT);
  ```

  Remember that IRIS GL color constants, such as BLACK, are not defined in OpenGL.
- Instead of using one of several differently named clear calls, you now clear several buffers with one call, **glClear()**, by ORing together buffer masks. For example, **czclear()** is replaced by

  ```
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT )
  ```
- IRIS GL respects the polygon stipple and the color write mask. OpenGL ignores the polygon stipple but respects the write mask. **czclear()** ignored both the polygon stipple and the write mask.

Table 3-4 lists the various clear calls with their IRIS GL equivalents.

Table 3-4 Calls for Clearing the Screen

IRIS GL Call	OpenGL Call	Meaning
acbuf(AC_CLEAR)	glClear(GL_ACCUM_BUFFER_BIT)	Clear the accumulation buffer.
—	glClearColor()	Set the RGBA clear color.
—	glClearIndex()	Set the clear color index.
clear()	glClear(GL_COLOR_BUFFER_BIT)	Clear the color buffer.
—	glClearDepth()	Specify the clear value for the depth buffer.
zclear()	glClear(GL_DEPTH_BUFFER_BIT)	Clear the depth buffer.
czclear()	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)	Clear the color buffer and the depth buffer.
—	glClearAccum()	Specify clear values for the accumulation buffer.
—	glClearStencil()	Specify the clear value for the stencil buffer.
sclear()	glClear(GL_STENCIL_BUFFER_BIT)	Clear the stencil buffer.

If your IRIS GL code used both **gclear()** and **sclear()**, you can instead use a single **glClear()** call. As a result, your program's performance may improve.

Porting Matrix and Transformation Calls

When porting matrix and transformation calls, consider the following issues:

- There is no single-matrix mode. OpenGL is always in double-matrix mode.
- Angles are now measured in degrees, instead of tenths of degrees.
- Projection matrix calls, like **glFrustum()** and **glOrtho()**, now multiply with the current matrix, instead of being loaded onto the current matrix.

- The OpenGL call **glRotate()** is different from **rotate()**. **glRotate()** lets you rotate around any arbitrary axis, instead of being confined to the *x*, *y*, and *z* axes. But you probably have to port **rotate()** calls explicitly, because *toogl* often doesn't translate them correctly. For example, *toogl* might translate

  ```
  rotate(200*(i+1), 'z');
  ```

  into

  ```
  glRotate(.1*(200*(i+1)), ( 'z')=='x', ( 'z')=='y',
                           ( 'z')=='z');
  ```

  toogl correctly switched to degrees from tenths of degrees, but didn't correctly handle the replacement of `z` with a vector for the z-axis. A better translation is

  ```
  glRotate(.1*(200*(i+1), 0.0, 0.0, 1.0);
  ```

- OpenGL documentation presents matrices in a manner more consistent with standard usage in linear algebra than did IRIS GL documentation. Specifically, in IRIS GL documentation, vectors are treated as rows, and a matrix is applied to a vector on the right of the vector. **multmatrix()** replaces the current matrix C with C′ = MC. In OpenGL documentation, vectors are treated as columns, and a matrix applies to a vector on the left of the vector. **glMultMatrix()** computes C′ = CM.

  A generic IRIS GL translation is shown in the equation in Figure 3-1.

$$[xyz1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{bmatrix} = [(x+Tx)(y+Ty)(z+Tz)1]$$

Figure 3-1 Generic IRIS GL Translation

A generic OpenGL translation is shown in the equation in Figure 3-2.

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+Tx \\ y+Ty \\ z+Tz \\ 1 \end{bmatrix}$$

Figure 3-2 Generic OpenGL Translation

The important thing is that this is a change in *documentation only*—OpenGL matrices are completely compatible with the ones in IRIS GL except that they are stored in column-major order. So, if you want the matrix shown in the equation in Figure 3-3 in your OpenGL application, you would declare it as follows:

```
float mat[16] = {a, e, i, m, b, f, j, n, c, g,
                 k, o, d, h, l, p}
```

$$\begin{bmatrix} a & b & c & d \\ a & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

Figure 3-3 OpenGL Matrix Example

- OpenGL has no equivalent to the **polarview()** call. You can replace such a call with a translation and three rotations. For example, the IRIS GL call

  ```
  polarview(distance, azimuth, incidence, twist);
  ```

  translates to

  ```
  glTranslatef( 0.0, 0.0, -distance);
  glRotatef( -twist * 10.0, 0.0, 0.0, 1.0);
  glRotatef( -incidence * 10.0, 1.0, 0.0, 0.0);
  glRotatef( -azimuth * 10.0, 0.0, 0.0, 1.0);
  ```

- The replacement for the **lookat()** call, **gluLookAt()**, takes an up vector instead of a twist angle. *toogl* doesn't translate this call correctly, so you have to port explicitly. See the gluLookAt reference page for more information.

Table 3-5 lists the OpenGL matrix calls and their IRIS GL equivalents.

Table 3-5 Matrix Operations

IRIS GL Call	OpenGL Call	Meaning
mmode()	glMatrixMode()	Set current matrix mode.
—	glLoadIdentity()	Replace current matrix with the identity matrix.
loadmatrix()	glLoadMatrixf(), glLoadMatrixd()	Replace current matrix with the specified matrix.

Table 3-5 (continued) Matrix Operations

IRIS GL Call	OpenGL Call	Meaning
multmatrix()	glMultMatrixf(), glMultMatrixd()	Post-multiply current matrix with the specified matrix (note that multmatrix() pre-multiplied).
mapw(), mapw2()	gluUnProject()	Project world space coordinates to object space (see also gluProject()).
ortho()	glOrtho()	Multiply current matrix by an orthographic projection matrix.
ortho2()	gluOrtho2D()	Define a two-dimensional orthographic projection matrix.
perspective()	gluPerspective()	Define a perspective projection matrix.
picksize()	gluPickMatrix()	Define a picking region.
popmatrix()	glPopMatrix()	Pop current matrix stack, replacing the current matrix with the one below it.
pushmatrix()	glPushMatrix()	Push current matrix stack down by one, duplicating the current matrix.
rotate(), rot()	glRotated(), glRotatef()	Rotate current coordinate system by the given angle about the vector from the origin through the given point. Note that rotate() rotated only about the *x*, *y*, and *z* axes.
scale()	glScaled(),glScalef()	Multiply current matrix by a scaling matrix.
translate()	glTranslatef(), glTranslated()	Move coordinate system origin to the specified point by multiplying the current matrix by a translation matrix.
window()	glFrustum()	Given coordinates for clipping planes, multiply the current matrix by a perspective matrix.

OpenGL has three matrix modes, which are set with **glMatrixMode()**. Table 3-6 lists the IRIS GL **mmode()** arguments in the first column and the corresponding arguments to **glMatrixMode()** in the second column.

Table 3-6 Matrix Modes

IRIS GL Matrix Mode	OpenGL Mode	Meaning	Min. Stack Depth
MTEXTURE	GL_TEXTURE	Operate on the texture matrix stack.	2
MVIEWING	GL_MODELVIEW	Operate on the modelview matrix stack.	32
MPROJECTION	GL_PROJECTION	Operate on the projection matrix stack.	2

Porting MSINGLE Mode Code

OpenGL has no equivalent for MSINGLE, single-matrix mode. Though use of this mode has been discouraged, it was the default for IRIS GL and your program may have used it. If it did, you have to reimplement part of it. OpenGL is always in double-matrix mode, and is initially in GL_MODELVIEW mode.

Most IRIS GL code in MSINGLE mode looks as follows:

```
...
projectionmatrix();
...
```

`projectionmatrix()` is one of the following: **ortho()**, **ortho2()**, **perspective()**, **window()**. To port to OpenGL, replace the MSINGLE mode **projectionmatrix()** call by the following pseudo-code:

```
...
glMatrixMode( GL_PROJECTION );
glLoadMatrix( identity matrix );
[one of these calls:
        glFrustrum(), glOrtho(), glOrtho2(), gluPerspective()];
glMatrixMode( GL_MODELVIEW );
glLoadMatrix( identity matrix );
```

Porting get* Calls for Matrices and Transformations

Table 3-7 maps IRIS GL matrix queries to OpenGL matrix queries.

Table 3-7 Arguments for Transformation Matrix Queries

IRIS GL Matrix Query	OpenGL glGet*() Matrix Query	Meaning
getmmode()	GL_MATRIX_MODE	Return the current matrix mode.
getmatrix() in MVIEWING mode	GL_MODELVIEW_MATRIX	Return a copy of the current modelview matrix.
getmatrix() in MPROJECTION mode	GL_PROJECTION_MATRIX	Return a copy of the current projection matrix.
getmatrix() in MTEXTURE mode	GL_TEXTURE_MATRIX	Return a copy of the current texture matrix.
—	GL_MAX_MODELVIEW_STACK_DEPTH	Return maximum supported depth of modelview matrix stack.
—	GL_MAX_PROJECTION_STACK_DEPTH	Return maximum supported depth of projection matrix stack.
—	GL_MAX_TEXTURE_STACK_DEPTH	Return maximum supported depth of texture matrix stack.
—	GL_MODELVIEW_STACK_DEPTH	Return number of matrices on modelview stack.
—	GL_PROJECTION_STACK_DEPTH	Return number of matrices on projection stack.
—	GL_TEXTURE_STACK_DEPTH	Return number of matrices on texture stack.

Porting Viewports, Screenmasks, and Scrboxes

The following IRIS GL calls have no direct OpenGL equivalent:

- **reshapeviewport()**
- **scrbox()**, **getscrbox()**

The IRIS GL **viewport()** call had as parameters the *x* coordinates (in pixels) for the left and right of the viewport rectangle and the *y* coordinates for the top and bottom. The OpenGL **glViewport()** call has as parameters the *x* and *y* coordinates (in pixels) of the lower left corner of the viewport rectangle, as well as the rectangle's width and height.

Table 3-8 lists the OpenGL equivalents for viewport commands.

Table 3-8 Viewport Calls

IRIS GL Call	OpenGL Call	Meaning
viewport(left, right, bottom, top)	glViewport(x, y, width, height)	Set the viewport.
popviewport() pushviewport()	glPopAttrib() glPushAttrib(GL_VIEWPORT_BIT)	Push and pop the stack.
getviewport()	glGet*(GL_VIEWPORT)	Return viewport dimensions.

Porting Clipping Planes

OpenGL implements clipping planes the way IRIS GL did, though you can now also query clipping planes. Table 3-9 lists the OpenGL equivalents to IRIS GL calls.

Table 3-9 Clipping Plane Calls

IRIS GL Call	OpenGL Call	Meaning
clipplane(*i*, CP_ON, *params*)	glEnable(GL_CLIP_PLANE*i*)	Enable clipping on plane *i.*
clipplane(*i*, CP_DEFINE, *plane*)	glClipPlane(GL_CLIP_PLANE*i*, *plane*)	Define clipping plane.
—	glGetClipPlane()	Return clipping plane equation.

Table 3-9 (continued) Clipping Plane Calls

IRIS GL Call	OpenGL Call	Meaning
—	glIsEnabled(GL_CLIP_PLANE*i*)	Return true if clip plane *i* is enabled.
scrmask()	glScissor()	Define the scissor box.
getscrmask()	glGet*(GL_SCISSOR_BOX)	Return the current scissor box.

To turn on the scissor test, call **glEnable()** with GL_SCISSOR_BOX as the parameter.

Porting Drawing Commands

The following sections discuss how to port IRIS GL drawing primitives, discussing the following topics:

Porting the IRIS GL Sphere Library

The sphere library that worked with IRIS GL isn't available for OpenGL. You can replace sphere library calls with quadrics routines from the GLU library or with the GLUT functions for geometric object rendering. Refer to the *OpenGL Programming Guide* and the

GLU reference pages in the *OpenGL Reference Manual* for details on using the GLU library. Table 3-10 summarizes OpenGL quadrics calls.

Table 3-10 Calls for Drawing Quadrics

OpenGL Call	Meaning
gluNewQuadric()	Create a new quadric object.
gluDeleteQuadric()	Delete a quadric object.
gluQuadricCallback()	Associate a callback with a quadric object, for error handling.
gluQuadricNormals()	Specify normals: no normals, one per face, or one per vertex.
gluQuadricOrientation()	Specify direction of normals: outward or inward.
gluQuadricTexture()	Turn texture coordinate generation on or off.
gluQuadricDrawstyle()	Specify drawing style: polygons, lines, points, and so on.
gluSphere()	Draw a sphere.
gluCylinder()	Draw a cylinder or cone.
gluPartialDisk()	Draw an arc.
gluDisk()	Draw a circle or disk.

You can use one quadric object for all quadrics you'd like to render in similar ways. The code fragment in Example 3-1 uses two quadrics objects to draw four quadrics, two of them textured.

Example 3-1 Drawing Quadrics Objects

```
GLUquadricObj    *texturedQuad, *plainQuad;

texturedQuad = gluNewQuadric(void);
gluQuadricTexture(texturedQuad, GL_TRUE);
gluQuadricOrientation(texturedQuad, GLU_OUTSIDE);
gluQuadricDrawStyle(texturedQuad, GLU_FILL);

plainQuad = gluNewQuadric(void);
gluQuadricDrawStyle(plainQuad, GLU_LINE);

glColor3f (1.0, 1.0, 1.0);
```

```
gluSphere(texturedQuad, 5.0, 20, 20);
glTranslatef(10.0, 10.0, 0.0);
gluCylinder(texturedQuad, 2.5, 5, 5, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluDisk(plainQuad, 2.0, 5.0, 10, 10);
glTranslatef(10.0, 10.0, 0.0);
gluSphere(plainQuad, 5.0, 20, 20);
```

Porting v() Commands

In IRIS GL, you use variations on the **v()** call to specify vertices. The OpenGL **glVertex()** call is a direct successor of this call:

```
glVertex2[d|f|i|s][v]( x, y );
glVertex3[d|f|i|s][v]( x, y, z);
glVertex4[d|f|i|s][v]( x, y, z, w);
```

glVertex() takes suffixes the same way other OpenGL calls do. The vector versions of the call take arrays of the proper size as arguments. In the 2D version, z = 0 and w = 1. In the 3D version, w = 1.

Porting bgn/end Commands

IRIS GL uses the begin/end paradigm but has a different call for each graphics primitive. For example, **bgnpolygon()** and **endpolygon()** draw polygons, and **bgnline()** and **endline()** draw lines. In OpenGL, you use the **glBegin()**/**glEnd()** structure. OpenGL draws most geometric objects by enclosing a series of calls that specify vertices, normals, textures, and colors between pairs of **glBegin()** and **glEnd()** calls.

```
void glBegin( GLenum mode) ;
   /* vertex list, colors, normals, textures, materials */
void glEnd( void );
```

glBegin() takes a single argument that specifies the drawing mode, and thus the primitive. Here's an OpenGL code fragment that draws a polygon and then a line:

```
glBegin( GL_POLYGON) ;
   glVertex2f(20.0, 10.0);
   glVertex2f(10.0, 30.0);
   glVertex2f(20.0, 50.0);
   glVertex2f(40.0, 50.0);
   glVertex2f(50.0, 30.0);
```

```
   glVertex2f(40.0, 10.0);
glEnd();

glBegin( GL_LINES ) ;
   glVertex2i(100,100);
   glVertex2i(500,500);
glEnd();
```

In OpenGL, you draw different geometric objects by specifying different arguments to **glBegin()**. These arguments are listed in Table 3-11 below, along with the IRIS GL calls they replace (if any). There is no limit to the number of vertices you can specify between a **glBegin()**/**glEnd()** pair.

Table 3-11 Calls for Drawing Primitives

IRIS GL Call	Value of glBegin() Mode	Meaning
bgnpoint()	GL_POINTS	Individual points
bgnline()	GL_LINE_STRIP	Series of connected line segments
bgnclosedline()	GL_LINE_LOOP	Series of connected line segments, with a segment added between first and last vertices
—	GL_LINES	Pairs of vertices interpreted as individual line segments
bgnpolygon()	GL_POLYGON	Boundary of a simple convex polygon
—	GL_TRIANGLES	Triples of vertices interpreted as triangles
bgntmesh()	GL_TRIANGLE_STRIP	Linked strips of triangles
—	GL_TRIANGLE_FAN	Linked fans of triangles
—	GL_QUADS	Quadruples of vertices interpreted as quadrilaterals
bgnqstrip()	GL_QUAD_STRIP	Linked strips of quadrilaterals

For a detailed discussion of the differences between triangle meshes, strips, and fans, see “Porting Triangles” on page 41.

In addition to specifying vertices inside a **glBegin()/glEnd()** pair, you can also specify a current normal, current texture coordinates, and a current color. Table 3-12 lists the commands valid inside a **glBegin()/glEnd()** pair.

Table 3-12 Valid Commands Inside a Begin/End Structure

IRIS GL Call	OpenGL Equivalent	Meaning
v2*(), v3*(), v4*()	glVertex*()	Set vertex coordinates.
RGBcolor(), cpack()	glColor*()	Set current color.
color(), colorf()	glIndex*()	Set current color index.
n3f()	glNormal*()	Set normal vector coordinates.
—	glEvalCoord()	Evaluate enabled one- and two-dimensional maps.
callobj()	glCallList(), glCallLists()	Execute display list(s).
t2()	glTexCoord()	Set texture coordinates.
—	glEdgeFlag()	Control drawing edges.
lmbind()	glMaterial()	Set material properties.

If you use any other OpenGL command inside a **glBegin()/glEnd()** pair, results are unpredictable and an error may result.

Porting Points

OpenGL has no command to draw a single point. Otherwise, porting point calls is straightforward. Table 3-13 lists commands for drawing points.

Table 3-13 Calls for Drawing Points

IRIS GL Call	OpenGL Equivalent	Meaning
pnt()	—	Draw a single point.
bgnpoint(), endpoint()	glBegin(GL_POINTS), glEnd()	Interpret vertices as points.

Table 3-13 (continued) Calls for Drawing Points

IRIS GL Call	OpenGL Equivalent	Meaning
pntsize()	glPointSize()	Set point size in pixels.
pntsmooth()	glEnable(GL_POINT_SMOOTH)	Turn on point antialiasing (see "Porting Antialiasing Calls" on page 58).

See the glPointSize reference page for information about related **glGet*()** commands.

Porting Lines

Porting code that draws lines is fairly straightforward, though you should note the differences in the way OpenGL does stipples.

Table 3-14 Calls for Drawing Lines

IRIS GL Call	OpenGL Call	Meaning
bgnclosedline(), endclosedline()	glBegin(GL_LINE_LOOP) glEnd()	Draw a closed line.
bgnline()	glBegin(GL_LINE_STRIP)	Draw line segments.
linewidth()	glLineWidth()	Set line width.
getlwidth()	glGet*(GL_LINE_WIDTH)	Return current line width.
deflinestyle() setlinestyle()	glLineStipple(*factor*, *pattern*)	Specify a line stipple pattern.
lsrepeat()	*factor* argument of glLineStipple()	Set a repeat factor for the line style.
getlstyle()	glGet*(GL_LINE_STIPPLE_PATTERN)	Return line stipple pattern.
getlsrepeat()	glGet*(GL_LINE_STIPPLE_REPEAT)	Return repeat factor.
linesmooth(), smoothline()	glEnable(GL_LINE_SMOOTH)	Turn on line antialiasing (see "Porting Antialiasing Calls" on page 58).

There are no tables for line stipples. OpenGL maintains only one line stipple pattern. You can use **glPushAttrib()** and **glPopAttrib()** to switch between different stipple patterns.

Old-style line style routines are not supported by OpenGL. If you used the calls: **draw()**, **lsbackup()**, **getlsbackup()**, **resetls()**, **getresetls()**, reimplement that part of your program.

For information on drawing antialiased lines, see "Porting Antialiasing Calls" on page 58.

Porting Polygons and Quadrilaterals

When porting polygons and quadrilaterals, consider the following issues:

- There is no direct equivalent for `concave(TRUE)`. Consider using the GLU tessellation routines described in "Porting Tessellated Polygons" on page 41.
- Polygon modes are now set differently.
- These older polygon drawing calls have no direct equivalents in OpenGL:
  - the **poly()** family of routines
  - the **polf()** family of routines
  - **pmv()**, **pdr()**, and **pclos()**
  - **rpmv()** and **rpdr()**
  - **splf()**
  - **spclos()**

  If you used these calls, reimplement that part of the program using `glBegin(GL_POLYGON)`.

Table 3-15 lists the OpenGL equivalents to IRIS GL polygon drawing calls.

Table 3-15 Calls for Drawing Polygons

IRIS GL Call	OpenGL Equivalent	Meaning
bgnpolygon(), endpolygon()	glBegin(GL_POLYGON), glEnd()	Vertices define boundary of a simple convex polygon.
—	glBegin(GL_QUADS), glEnd()	Interpret quadruples of vertices as quadrilaterals.
bgnqstrip(), endqstrip()	glBegin(GL_QUAD_STRIP), glEnd()	Interpret vertices as linked strips of quadrilaterals.

Table 3-15 Calls for Drawing Polygons

IRIS GL Call	OpenGL Equivalent	Meaning
—	glEdgeFlag()	
polymode()	glPolygonMode()	Set polygon drawing mode.
rect(), rectf()	glRect()	Draw a rectangle.
sbox(), sboxf()	—	Draw a screen-aligned rectangle.

Setting Polygon Modes

The call for setting the polygon mode has changed slightly. The OpenGL call **glPolygonMode()** allows you to specify which side of a polygon (front or back) the mode applies to. Its syntax is

```
void glPolygonMode( GLenum face, GLenum mode )
```

face is one of the following:

GL_FRONT	Mode applies to front-facing polygons.
GL_BACK	Mode applies to back-facing polygons.
GL_FRONT_AND_BACK	Mode applies to both front- and back-facing polygons.

The equivalents to IRIS GL **polymode()** calls would use GL_FRONT_AND_BACK. Table 3-16 lists IRIS GL polygon modes and the corresponding OpenGL modes.

Table 3-16 Polygon Modes

IRIS GL Mode	OpenGL Mode	Meaning
PYM_POINT	GL_POINT	Draw vertices as points.
PYM_LINE	GL_LINE	Draw boundary edges as line segments.
PYM_FILL	GL_FILL	Draw polygon interior filled.
PYM_HOLLOW	—	Fill only interior pixels at the boundaries.

Setting Polygon Stipples

When porting polygon stipples, consider the following issues:

- There are no tables for polygon stipples. OpenGL keeps only one stipple pattern. You can use display lists to store different stipple patterns.
- The polygon stipple bitmap size is always a 32 x 32 bit pattern.
- Stipple encoding is affected by **glPixelStore()**. See "Porting Pixel Operations" on page 46 for more information.

Table 3-17 lists polygon stipple calls.

Table 3-17 Polygon Stipple Calls

IRIS GL Call	OpenGL Call	Meaning
defpattern()	glPolygonStipple()	Set the stipple pattern.
setpattern()	—	OpenGL keeps only one polygon stipple pattern.
getpattern()	glGetPolygonStipple()	Return the stipple bitmap (used to return an index).

Enable and disable polygon stippling by passing GL_POLYGON_STIPPLE as an argument to **glEnable()** and **glDisable()**.

Example 3-2 shows an OpenGL code fragment that demonstrates polygon stippling.

Example 3-2 OpenGL Polygon Stippling

```
/*  polys.c  */
#include <GL/gl.h>
#include <GL/glu.h>

void display(void)
{
    GLubyte fly[] = {
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
      0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
      0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
      0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
```

```
      0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
      0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
      0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
      0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
      0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
      0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
      0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
      0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
      0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08
    };
    GLubyte halftone[] = {
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
      0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55
    };

    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0); /* draw all polys in white */

/*  draw 1 solid unstippled rectangle, then 2 stippled ones*/
    glRectf (25.0, 25.0, 125.0, 125.0);
    glEnable (GL_POLYGON_STIPPLE);
    glPolygonStipple (fly);
    glRectf (125.0, 25.0, 225.0, 125.0);
    glPolygonStipple (halftone);
    glRectf (225.0, 25.0, 325.0, 125.0);
    glDisable (GL_POLYGON_STIPPLE);
    glFlush ();
}
```

Porting Tessellated Polygons

The GLU has routines you can use to draw concave polygons. You no longer just use **concave(TRUE)** and then **bgnpolygon()**.

To draw a concave polygon with OpenGL, follow these steps:

1. Create a tesselation object.
2. Define callbacks that will be used to process the triangles generated by the tessellator.
3. Specify the concave polygon to be tessellated.

Table 3-18 lists the calls for drawing tessellated polygons.

Table 3-18 Tessellated Polygon Calls

GLU Call	Meaning
gluNewTess()	Create a new tessellation object.
gluDeleteTess()	Delete a tessellation object.
gluTessCallback()	—
gluBeginPolygon()	Begin the polygon specification.
gluTessVertex()	Specify a polygon vertex. Successive gluTessVertex() calls describe a closed contour.
gluNextContour()	Indicate that the next series of vertices describe a new contour.
gluEndPolygon()	End the polygon specification.

For details, see the reference pages for the commands in Table 3-18.

Porting Triangles

OpenGL provides three ways to draw triangles: separate triangles, triangle strips, and triangle fans.

When porting triangles, consider the following issues:

- There's no OpenGL equivalent for **swaptmesh()**. Instead, use a combination of triangles, triangle strips, and triangle fans.
- If your IRIS GL program draws individual triangles by surrounding each triangle with a **bgntmesh()** / **endtmesh()** pair, surround the entire group of individual triangles with just one `glBegin(GL_TRIANGLES)` / `glEnd()` pair in your OpenGL program, for a noticeable performance increase.

Table 3-19 lists the commands for drawing triangles.

Table 3-19 Calls for Drawing Triangles

IRIS GL Call	Equivalent glBegin() Argument	Meaning
—	GL_TRIANGLES	Triples of vertices interpreted as triangles.
bgntmesh(), endtmesh()	GL_TRIANGLE_STRIP	Linked strips of triangles.
—	GL_TRIANGLE_FAN	Linked fans of triangles.

Porting Arcs and Circles

In OpenGL, filled arcs and circles are drawn with the same calls as unfilled arcs and circles. See the reference pages for specifics. Table 3-20 lists the IRIS GL arc and circle commands and the corresponding OpenGL (GLU) commands.

Table 3-20 Calls for Drawing Arcs and Circles

IRIS GL Call	OpenGL Call	Meaning
arc(), arcf()	gluPartialDisk()	Draw an arc.
circ(), circf()	gluDisk()	Draw a circle or disk.

The **gluPartialDisk()** call is different from the **arc()** call. See the gluPartialDisk reference page for complete information.

IRIS GL arcs and circles are called disks and partial disks in OpenGL. You can do some things with OpenGL disks and partial disks that you could not do with IRIS GL. See the

OpenGL Programming Guide and the reference pages in the *OpenGL Reference Manual* for detailed information.

When porting arcs and circles, consider these issues:

- Angles are no longer measured in tenths of degrees, but simply in degrees.
- The start angle is measured from the positive *y* axis, and not from the *x* axis.
- The sweep angle is now clockwise instead of counterclockwise, as shown in Figure 3-4.

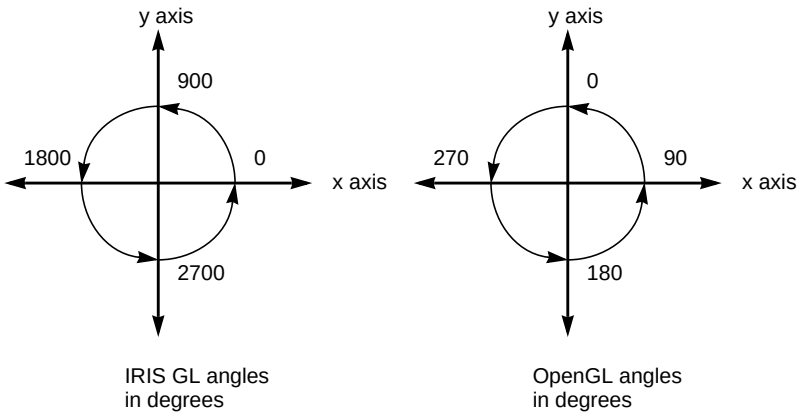

Figure 3-4 Drawing Angles: Comparing IRIS GL and OpenGL

Porting Spheres

When porting spheres, consider these issues:

- In OpenGL, you cannot control the type of primitives used to draw the sphere. Instead, you can control drawing precision by using the *slices* and *stacks* parameters. Slices are longitudinal; stacks are latitudinal.
- Spheres are now drawn centered at the origin. Instead of specifying the location, as you used to in **sphdraw()** calls, precede a **gluSphere()** call with a translation.
- The sphere library isn't yet available for OpenGL—see "Porting the IRIS GL Sphere Library" on page 31 for more information about replacing sphere library calls.

Table 3-21 lists the IRIS GL calls for drawing spheres along with the corresponding GLU calls where available.

Table 3-21 Calls for Drawing Spheres

IRIS GL Call	GLU Call	Notes
sphobj()	gluNewQuadric()	Create a new sphere object.
sphfree()	gluDeleteQuadric()	Delete sphere object and free memory used.
sphdraw()	gluSphere()	Draw a sphere.
sphmode()	—	Set sphere attributes.
sphrotmatrix()	—	Control sphere orientation.
sphgnpolys()	—	Return number of polygons in current sphere.

Porting Color, Shading, and Writemask Commands

When porting color, shading, and writemask calls, note that color map implementation differs between OpenGL and IRIS GL and consider these issues:

- Although you can set color map indices with the OpenGL **glIndex()** call, OpenGL doesn't provide a routine for loading color map indices. See "Using X Color Maps" on page 105 for an example code fragment that sets up a color map.
- Color values are normalized to their data type. See the glColor reference page for details.
- There is no simple equivalent for **cpack()**. You can use **glColor()** instead, but you have to port explicitly.
- Some calls to **c()** or **color()** may have to be translated to **glClearColor()** or **glClearIndex()** and not **glColor()** or **glIndex()**. See "Porting Screen and Buffer Clearing Commands" on page 23 for details.
- The RGBA writemask is not for each bit, just for each component.
- IRIS GL provided defined color constants: BLACK, BLUE, RED, GREEN, MAGENTA, CYAN, YELLOW, and WHITE. OpenGL doesn't provide these constants and *toogl* doesn't translate them, so you have to port them explicitly.

Porting Color Calls

Table 3-22 lists IRIS GL color calls and their OpenGL equivalents.

Table 3-22 Color Calls

IRIS GL Call	OpenGL Call	Meaning
c3*(), c4*()	glColor*()	Sets RGB color.
color(), colorf()	glIndex*()	Sets the color index.
getcolor()	glGet*(GL_CURRENT_INDEX)	Returns the current color index.
getmcolor()	XQueryColor()	Gets a copy of a colormap entry's RGB values.
gRGBcolor()	glGet*(GL_CURRENT_COLOR)	Gets the current RGB color values.
mapcolor()	XStoreColor()	See "Using X Color Maps" on page 105.
RGBcolor()	glColor()	Sets RGB color.
writemask()	glIndexMask()	Sets the color index mode color mask.
wmpack() RGBwritemask()	glColorMask()	Sets the RGB color mode mask.
getwritemask()	glGet*(GL_COLOR_WRITEMASK) glGet*(GL_INDEX_WRITEMASK)	Gets the color mask.
gRGBmask()	glGet*(GL_COLOR_WRITEMASK)	Gets the color mask.
zwritemask()	glDepthMask()	—

Note: Be careful when replacing **zwritemask()** with **glDepthMask()**: **glDepthMask()** takes a boolean argument; **zwritemask()** takes a bitfield.

If you want to use multiple color maps, use the X colormap facilities. The functions **multimap()**, **onemap()**, **getcmmode()**, **setmap()**, and **getmap()** have no OpenGL equivalents.

Porting Shading Models

Just like IRIS GL, OpenGL lets you switch between smooth (Gouraud) shading and flat shading. Table 3-23 lists the calls.

Table 3-23 Shading and Dithering

IRIS GL Call	OpenGL Call	Meaning
shademodel(FLAT)	glShadeModel(GL_FLAT)	Do flat shading.
shademodel(GOURAUD)	glShadeModel(GL_SMOOTH)	Do smooth shading.
getsm()	glGet*(GL_SHADE_MODEL)	Return current shade model.
dither(DT_ON) dither(DT_OFF)	glEnable(GL_DITHER) glDisable(GL_DITHER)	Turn dithering on/off.

Smooth shading and dithering are on by default, as in IRIS GL.

Porting Pixel Operations

When porting pixel operations, consider the following issues:

- Logical pixel operations are not applied to RGBA color buffers. See the glLogicOp reference page for more information.
- In general, IRIS GL used the ABGR format for pixels (that is, with color components in the order Alpha, Blue, Green, Red), while OpenGL uses the RGBA format. Although **glPixelStore()** can reverse the order of bytes within a color component, it can't reverse the order of the components within a pixel; thus, it can't be used to convert IRIS GL pixels to OpenGL pixels. Instead, you must reverse the order of the components yourself.
- When porting **lrectwrite()** calls, be careful to note where **lrectwrite()** is writing (for instance, it could be writing to the depth buffer).
- If you wanted to read from the *z*-buffer in IRIS GL, you specified that buffer with **readsource()** and then used **lrectread()** or **rectread()** to do the reading. If you want to read from the *z*-buffer in OpenGL, you simply specify that buffer as a parameter to **glReadPixels()**.

OpenGL provides some additional flexibility in pixel operations. Table 3-24 lists calls for pixel operations.

Table 3-24 Pixel Operations

IRIS GL Call	OpenGL Call	Meaning
lrectread(), rectread(), readRGB()	glReadPixels()	Read a block of pixels from the frame buffer.
lrectwrite(), rectwrite()	glDrawPixels()	Write a block of pixels to the frame buffer.
rectcopy()	glCopyPixels()	Copy pixels in the frame buffer.
rectzoom()	glPixelZoom()	Specify pixel zoom factors for glDrawPixels() and glCopyPixels().
cmov()	glRasterPos()	Specify raster position for pixel operations.
readsource()	glReadBuffer()	Select a color buffer source for pixels.
pixmode()	glPixelStore()	Set pixel storage modes.
pixmode()	glPixelTransfer()	Set pixel transfer modes.
logicop()	glLogicOp()	Specify a logical operation for pixel writes.
—	glEnable(GL_LOGIC_OP)	Turn on pixel logic operations.

See the reference page for glLogicOp for a list of possible logical operations.

Here's a code fragment that shows a typical pixel write operation:

```
unsigned long *packedRaster;
...
packedRaster[k] = 0x00000000;
...
lrectwrite(0, 0, xSize, ySize, packedRaster);
```

Here is how *toogl* translates the call to **lrectwrite()**:

```
/* OGLXXX lrectwrite: see man page for glDrawPixels */
glRasterPos2i(0,  0);
glDrawPixels(( xSize)-(0)+1, ( ySize)-( 0)+1, GL_RGBA,
             GL_UNSIGNED_BYTE, packedRaster);
```

After some tweaking, the finished code might look like this:

```
glRasterPos2i(0, 0);
glDrawPixels(xSize + 1, ySize + 1, GL_RGBA,
             GL_UNSIGNED_BYTE, packedRaster);
```

Porting Depth Cueing and Fog Commands

When porting depth cueing and fog commands, consider these issues:

- The fog calls have been restructured, so you have to rewrite them explicitly in most cases. The IRIS GL call **fogvertex()** set a mode and parameters affecting that mode. In OpenGL, you call **glFog()** once to set the mode, then again twice or more to set various parameters.
- Depth cueing is no longer a separate feature. Use linear fog instead of depth cueing. (This section provides an example of how to do this.) The following calls therefore have no direct OpenGL equivalent:
  - **depthcue()**
  - **lRGBrange()**
  - **lshaderange()**
  - **getdcm()**
- To adjust fog quality, call `glHint(GL_FOG_HINT)`.

Table 3-25 lists the IRIS GL calls for managing fog along with the corresponding OpenGL calls.

Table 3-25 Calls for Managing Fog

IRIS GL Call	OpenGL Call	Meaning
fogvertex()	glFog()	Set various fog parameters.
fogvertex(FG_ON)	glEnable(GL_FOG)	Turn fog on.
fogvertex(FG_OFF)	glDisable(GL_FOG)	Turn fog off.
depthcue()	glFog(GL_FOG_MODE, GL_LINEAR)	Use linear fog for depth cueing.

Table 3-26 lists the arguments you can pass to **glFog()**.

Table 3-26 Fog Parameters

Fog Parameter	Meaning	Default
GL_FOG_DENSITY	Fog density.	1.0
GL_FOG_START	Near distance for linear fog.	0.0
GL_FOG_END	Far distance for linear fog.	1.0
GL_FOG_INDEX	Fog color index.	0.0
GL_FOG_COLOR	Fog RGBA color.	(0, 0, 0, 0)
GL_FOG_MODE	Fog mode.	see Table 3-27

The OpenGL fog density argument differs from the IRIS GL fog density argument. They are related as follows:

- If *fogMode* is EXP2:

  openGLfogDensity = (IRISGLfogDensity) (sqrt(- log(1 / 255)))

- If *fogMode* is EXP:

  openGLfogDensity = (IRISGLfogDensity) (- log(1 / 255))

where

sqrt	Is the square root operation.
log	Is the natural logarithm.
IRISGLfogDensity	Is the IRIS GL fog density.
openGLfogDensity	Is the OpenGL fog density.

To switch between calculating fog in per-pixel mode and per-vertex mode, use `glHint(GL_FOG_HINT,` *hintMode*`)`. Two hint modes are available:

GL_NICEST	per-pixel fog calculation
GL_FASTEST	per-vertex fog calculation

Table 3-27 lists the OpenGL equivalents for IRIS GL fog modes.

Table 3-27 Fog Modes

IRIS GL Fog Mode	OpenGL Fog Mode	Hint Mode	Meaning
FG_VTX_EXP, FG_PIX_EXP	GL_EXP	GL_FASTEST, GL_NICEST	Heavy fog mode (default)
FG_VTX_EXP2, FG_PIX_EXP2	GL_EXP2	GL_FASTEST, GL_NICEST	Haze mode
FG_VTX_LIN, FG_PIX_LIN	GL_LINEAR	GL_FASTEST, GL_NICEST	Linear fog mode (use for depthcueing)

Example 3-3 shows a code fragment that demonstrates depth cueing in OpenGL.

Example 3-3 Depth Cueing in OpenGL

```
/*
 *  depthcue.c
 *  This program draws a wireframe model, which uses
 *  intensity (brightness) to give clues to distance.
 *  Fog is used to achieve this effect.
 */
#include <stdlib.h>
#include <GL/glut.h>

/*  Initialize linear fog for depth cueing.
 */
void myinit(void)
{
    GLfloat fogColor[4] = {0.0, 0.0, 0.0, 1.0};

    glEnable(GL_FOG);
    glFogi(GL_FOG_MODE, GL_LINEAR);
    glHint(GL_FOG_HINT, GL_NICEST);  /*  per pixel   */
    glFogf(GL_FOG_START, 3.0);
    glFogf(GL_FOG_END, 5.0);
    glFogfv(GL_FOG_COLOR, fogColor);
    glClearColor(0.0, 0.0, 0.0, 1.0);

    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);
```

```
    glShadeModel(GL_FLAT);
}

/*  display() draws an icosahedron.
 */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glutWireIcosahedron();
    glFlush();
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, (GLfloat)w/(GLfloat)h, 3.0, 5.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -4.0);  /*  move object into view   */
}

/*  Main Loop
 */
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
    glutCreateWindow(argv[0]);
    myinit();
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;             /* ANSI C requires main to return int. */
}
```

Porting Curve and Surface Commands

OpenGL doesn't support equivalents to the old-style curves and surface patches. You have to reimplement your code if it uses any of these calls:

- **defbasis()**
- **curvebasis()**, **curveprecision()**, **crv()**, **crvn()**, **rcrv()**, **rcrvn()**, and **curveit()**
- **patchbasis()**, **patchcurves()**, **patchprecision()**, **patch()**, and **rpatch()**

Silicon Graphics recommends that you reimplement these calls using evaluators, rather than trying to replace them with NURBS. Refer to the *OpenGL Reference Manual* and to the section "Evaluators" on page 440 of the *OpenGL Programming Guide, Second Edition*, for more information on using evaluators.

NURBS Objects

OpenGL treats NURBS as objects, similar to the way it treats quadrics: you create a NURBS object and then specify how it should be rendered. Table 3-28 lists the NURBS object commands.

Table 3-28 Calls for Managing NURBS Objects

OpenGL Call	Meaning
gluNewNurbsRenderer()	Create a new NURBS object.
gluDeleteNurbsRenderer()	Delete a NURBS object.
gluNurbsCallback()	Associate a callback for error handling with a NURBS object.

When using NURBS objects, consider the following issues:

- NURBS control points are now floats, not doubles.
- The *stride* parameter is now counted in floats, not bytes.
- If you're using lighting and you're not specifying normals, call **glEnable()** with GL_AUTO_NORMAL as the parameter to generate normals automatically.

NURBS Curves

The OpenGL calls for drawing NURBS are similar to the IRIS GL calls. You specify knot sequences and control points using a **gluNurbsCurve()** call, which must be contained within a **glBeginCurve()/glEndCurve()** pair.

Table 3-29 summarizes the calls for drawing NURBS curves.

Table 3-29 Calls for Drawing NURBS Curves

IRIS GL Call	OpenGL Call	Meaning
bgncurve()	gluBeginCurve()	Begin a curve definition.
nurbscurve()	gluNurbsCurve()	Specify curve attributes.
endcurve()	gluEndCurve()	End a curve definition.

Position, texture, and color coordinates are associated by presenting each as a separate **gluNurbsCurve()** inside the begin/end pair. You can make no more than one call to **gluNurbsCurve()** for each piece of color, position, and texture data within a single **gluBeginCurve()/gluEndCurve()** pair. You must make exactly one call to describe the position of the curve (a GL_MAP1_VERTEX_3 or GL_MAP1_VERTEX_4 description). When you call **gluEndCurve()**, the curve will be tessellated into line segments and then rendered.

Table 3-30 lists NURBS curve types.

Table 3-30 NURBS Curve Types

IRIS GL Type	OpenGL Type	Meaning
N_V3D	GL_MAP1_VERTEX_3	Polynomial curve.
N_V3DR	GL_MAP1_VERTEX_4	Rational curve.
—	GL_MAP1_TEXTURE_COORD_*	Control points are texture coordinates.
—	GL_MAP1_NORMAL	Control points are normals.

For more information on available evaluator types, see the glMap1 reference page.

Trimming Curves

OpenGL trimming curves are similar to IRIS GL trimming curves. Table 3-31 lists the calls for defining trimming curves.

Table 3-31 Calls for Drawing NURBS Trimming Curves

IRIS GL Call	OpenGL Call	Meaning
bgntrim()	gluBeginTrim()	Begin trimming curve definition.
pwlcurve()	gluPwlCurve()	Define a piecewise linear curve.
nurbscurve()	gluNurbsCurve()	Specify trimming curve attributes.
endtrim()	gluEndTrim()	End trimming curve definition.

NURBS Surfaces

Table 3-32 summarizes the calls for drawing NURBS surfaces.

Table 3-32 Calls for Drawing NURBS Surfaces

IRIS GL Call	OpenGL Call	Meaning
bgnsurface()	gluBeginSurface()	Begin a surface definition.
nurbssurface()	gluNurbsSurface()	Specify surface attributes.
endsurface()	gluEndSurface()	End a surface definition.

Table 3-33 lists parameters for surface types.

Table 3-33 NURBS Surface Types

IRIS GL Type	OpenGL Type	Meaning
N_V3D	GL_MAP2_VERTEX_3	Polynomial curve
N_V3DR	GL_MAP2_VERTEX_4	Rational curve
N_C4D	GL_MAP2_COLOR_4	Control points define color surface in (R,G,B,A) form

Table 3-33 (continued) NURBS Surface Types

IRIS GL Type	OpenGL Type	Meaning
N_C4DR	—	—
N_T2D	GL_MAP2_TEXTURE_COORD_2	Control points are texture coordinates.
N_T2DR	GL_MAP2_TEXTURE_COORD_3	Control points are texture coordinates.
—	GL_MAP2_NORMAL	Control points are normals.

For more information on available evaluator types, see the glMap2 reference page.

Example 3-4 draws a trimmed NURBS surface.

Example 3-4 Drawing an OpenGL NURBS surface

```
/*
 *  trim.c
 *  This program draws a NURBS surface in the shape of a
 *  symmetrical hill, using both a NURBS curve and pwl
 *  (piecewise linear) curve to trim part of the surface.
 */
#include <stdlib.h>
#include <GL/glut.h>
#include <stdio.h>

GLfloat ctlpoints[4][4][3];

GLUnurbsObj *theNurb;

/*
 *  Initializes the control points of the surface to a small hill.
 *  The control points range from -3 to +3 in x, y, and z
 */
void init_surface(void)
{
   int u, v;

   for (u = 0; u < 4; u++) {
      for (v = 0; v < 4; v++) {
         ctlpoints[u][v][0] = 2.0 * ((GLfloat)u - 1.5);
         ctlpoints[u][v][1] = 2.0 * ((GLfloat)v - 1.5);

         if ((u == 1 || u == 2) && (v == 1 || v == 2))
```

```
            ctlpoints[u][v][2] = 3.0;
         else
            ctlpoints[u][v][2] = -3.0;
      }
   }
}

void nurbsError(GLenum errorCode)
{
   const GLubyte *estring;

   estring = gluErrorString(errorCode);
   fprintf (stderr, "Nurbs Error: %s\n", estring);
   exit (0);
}

/*  Initialize material property and depth buffer.
 */
void init(void)
{
   GLfloat mat_diffuse[] = { 0.7, 0.7, 0.7, 1.0 };
   GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
   GLfloat mat_shininess[] = { 100.0 };

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
   glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   glEnable(GL_AUTO_NORMAL);
   glEnable(GL_NORMALIZE);

   init_surface();

   theNurb = gluNewNurbsRenderer();
   gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
   gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
   gluNurbsCallback(theNurb, GLU_ERROR, (GLvoid
(CALLBACK*)())nurbsError);
}

void display(void)
```

```
{
   GLfloat knots[8] = {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   GLfloat edgePt[5][2] = /* counter clockwise */
      {{0.0, 0.0}, {1.0, 0.0}, {1.0, 1.0}, {0.0, 1.0}, {0.0, 0.0}};
   GLfloat curvePt[4][2] = /* clockwise */
      {{0.25, 0.5}, {0.25, 0.75}, {0.75, 0.75}, {0.75, 0.5}};
   GLfloat curveKnots[8] =
      {0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0};
   GLfloat pwlPt[4][2] = /* clockwise */
      {{0.75, 0.5}, {0.5, 0.25}, {0.25, 0.5}};

   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
   glRotatef(330.0, 1.,0.,0.);
   glScalef(0.5, 0.5, 0.5);

   gluBeginSurface(theNurb);
   gluNurbsSurface(theNurb, 8, knots, 8, knots,
                   4*3, 3, &ctlpoints[0][0][0],
                   4, 4, GL_MAP2_VERTEX_3);
   gluBeginTrim(theNurb);
      gluPwlCurve(theNurb, 5, &edgePt[0][0], 2, GLU_MAP1_TRIM_2);
   gluEndTrim(theNurb);
   gluBeginTrim(theNurb);
      gluNurbsCurve(theNurb, 8, curveKnots, 2,
                    &curvePt[0][0], 4, GLU_MAP1_TRIM_2);
      gluPwlCurve(theNurb, 3, &pwlPt[0][0], 2, GLU_MAP1_TRIM_2);
   gluEndTrim(theNurb);
   gluEndSurface(theNurb);

   glPopMatrix();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei)w, (GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0, (GLdouble)w/(GLdouble)h, 3.0, 8.0);

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -5.0);
}
```

```
/* ARGSUSED1 */
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}

/*  Main Loop
 */
int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutReshapeFunc(reshape);
   glutDisplayFunc(display);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Porting Antialiasing Calls

This section discusses topics related to antialiasing:

- "Blending"
- "afunction() Test Functions" on page 60
- "Antialiasing Calls" on page 60

In OpenGL, subpixel mode is always on, so the IRIS GL call `subpixel(TRUE)` is not necessary and has no OpenGL equivalent.

Blending

Blending is off by default. If you use _DA or _MDA blend functions, you have to allocate destination alpha bits when you choose a visual. You have to use X to choose the visual, so refer to Chapter 4.

Tip: In IRIS GL, when drawing to both front and back buffers, blending is done by reading *one* of the buffers, blending with that color, and then writing the result to both buffers. In OpenGL, however, each buffer is read in turn, blended, and then written.

Table 3-34 lists IRIS GL and OpenGL blending calls.

Table 3-34 Blending Calls

IRIS GL	OpenGL	Meaning
—	glEnable(GL_BLEND)	Turn on blending.
blendfunction()	glBlendFunc()	Specify a blend function.

The calls **glBlendFunc()** and **blendfunction()** are almost identical. Table 3-35 lists the OpenGL equivalents to the IRIS GL blend factors.

Table 3-35 Blending Factors

IRIS GL	OpenGL	Notes
BF_ZERO	GL_ZERO	
BF_ONE	GL_ONE	
BF_SA	GL_SRC_ALPHA	
BF_MSA	GL_ONE_MINUS_SRC_ALPHA	
BF_DA	GL_DST_ALPHA	
BF_MDA	GL_ONE_MINUS_DST_ALPHA	
BF_SC	GL_SRC_COLOR	
BF_MSC	GL_ONE_MINUS_SRC_COLOR	Destination only
BF_DC	GL_DST_COLOR	Source only

Table 3-35 (continued) Blending Factors

IRIS GL	OpenGL	Notes
BF_MDC	GL_ONE_MINUS_DST_COLOR	Source only
BF_MIN_SA_MDA	GL_SRC_ALPHA_SATURATE	

afunction() Test Functions

Table 3-36 lists the available alpha test functions.

Table 3-36 Alpha Test Functions

afunction()	glAlphaFunc()
AF_NOTEQUAL	GL_NOTEQUAL
AF_ALWAYS	GL_ALWAYS
AF_NEVER	GL_NEVER
AF_LESS	GL_LESS
AF_EQUAL	GL_EQUAL
AF_LEQUAL	GL_LEQUAL
AF_GREATER	GL_GREATER
AF_GEQUAL	GL_GEQUAL

Antialiasing Calls

OpenGL has direct equivalents to the IRIS GL antialiasing calls. Table 3-37 lists them.

Table 3-37 Calls to Draw Antialiased Primitives

IRIS GL Call	OpenGL Call	Meaning
pntsmooth()	glEnable(GL_POINT_SMOOTH)	Enable antialiasing of points.
linesmooth()	glEnable(GL_LINE_SMOOTH)	Enable antialiasing of lines.
polysmooth()	glEnable(GL_POLYGON_SMOOTH)	Enable antialiasing of polygons.

Use the corresponding **glDisable()** calls to turn off antialiasing.

In IRIS GL, you can control the quality of the antialiasing by calling

```
linesmooth(SML_ON + SML_SMOOTHER);
```

OpenGL provides similar control—use **glHint()**:

```
glHint(GL_POINT_SMOOTH_HINT, hintMode);
glHint(GL_LINE_SMOOTH_HINT, hintMode);
glHint(GL_POLYGON_SMOOTH_HINT, hintMode);
```

hintMode is one of the following:

GL_NICEST	Use the highest quality smoothing.
GL_FASTEST	Use the most efficient smoothing.
GL_DONT_CARE	You don't care which smoothing method is used.

You could perform end correction in IRIS GL by calling

```
linesmooth(SML_ON + SML_END_CORRECT);
```

OpenGL doesn't provide an equivalent for this call.

Accumulation Buffer Calls

You must allocate your accumulation buffer by requesting the appropriate visual with **glXChooseVisual()**. For information on **glXChooseVisual()**, see the glXIntro and glXChooseVisual reference pages and refer to Chapter 4.

IRIS GL allows you to draw colors in the depth buffer, so **acbuf()** can use that buffer as a color source for accumulation. Some developers have used this depth-buffer reading capability to put depth data into accumulation buffers as well. OpenGL, on the other hand, doesn't put color information in the depth buffer; **glAccum()** thus can't read any information from the depth buffer.

To emulate accumulation from the depth buffer (using a configuration that supports auxiliary buffers) use the following procedure:

1. Use **glReadPixels()** to read from the depth buffer.
2. Massage the results as necessary.

3. Draw the resulting data to an auxiliary buffer.
4. Select this auxiliary buffer with **glReadBuffer()**, and use **glAccum()** to accumulate from that buffer.

This procedure requires caution in converting among data types.

Except as noted above, porting accumulation buffer calls is straightforward. Table 3-38 lists calls that affect the accumulation buffer.

Table 3-38 Accumulation Buffer Calls

IRIS GL Call	OpenGL Call	Meaning
acbuf()	glAccum()	Operate on the accumulation buffer.
—	glClearAccum()	Set clear values for accumulation buffer.
acbuf(AC_CLEAR)	glClear(GL_ACCUM_BUFFER_BIT)	Clear the accumulation buffer.
acsize()	glXChooseVisual()	Specify number of bitplanes per color component in accumulation buffer.

Table 3-39 lists the IRIS GL **acbuf()** arguments along with the corresponding arguments to the OpenGL **glAccum()** call.

Table 3-39 Accumulation Buffer Operations

IRIS GL Argument	OpenGL Argument
AC_ACCUMULATE	GL_ACCUM
AC_CLEAR_ACCUMULATE	GL_LOAD
AC_RETURN	GL_RETURN
AC_MULT	GL_MULT
AC_ADD	GL_ADD

Stencil Plane Calls

In OpenGL, you allocate stencil planes by requesting the appropriate visual with **glXChooseVisual()**. (For information on **glXChooseVisual()**, see the glXIntro and glXChooseVisual reference pages and refer to Chapter 4.) Otherwise, porting should be straightforward. Table 3-40 lists calls that affect the stencil planes.

Table 3-40 Stencil Operations

IRIS GL Call	OpenGL Call	Meaning
stensize()	glXChooseVisual()	—
stencil(TRUE, ...)	glEnable(GL_STENCIL_TEST)	Enable stencil tests.
stencil()	glStencilOp()	Set stencil test actions.
stencil(... func, ...)	glStencilFunc()	Set function & reference value for stencil testing.
swritemask()	glStencilMask()	Specify which stencil bits can be written.
—	glClearStencil()	Specify the clear value for the stencil buffer.
sclear()	glClear(GL_STENCIL_BUFFER_BIT)	—

Stencil comparison functions and stencil pass/fail operations are almost equivalent in OpenGL and IRIS GL. The IRIS GL stencil function flags are prefaced with SF, the OpenGL flags with GL. IRIS GL pass/fail operation flags are prefaced with ST, the OpenGL flags with GL. Compare the IRIS GL and OpenGL reference pages for further details.

Porting Display Lists

The OpenGL implementation of display lists is similar to the IRIS GL implementation, with two exceptions: you can't edit display lists once you've created them and you can't call functions from within display lists.

Because you can't edit or call functions from within display lists, these IRIS GL commands have no equivalents in OpenGL:

- **editobj()**
- **objdelete()**, **objinsert()**, and **objreplace()**
- **maketag()**, **gentag()**, **istag()**, and **deltag()**
- **callfunc()**

In IRIS GL, you used the commands **makeobj()** and **closeobj()** to create display lists. In OpenGL, you use **glNewList()** and **glEndList()**. For details on using **glNewList()** (including a description of the two list modes and a list of commands that are not compiled into the display list but are executed immediately), see the glNewList reference page and the *OpenGL Programming Guide*.

Table 3-41 lists the IRIS GL display list commands with the corresponding OpenGL commands.

Table 3-41 Display List Commands

IRIS GL Call	OpenGL Call	Meaning
makeobj()	glNewList()	Create a new display list.
closeobj()	glEndList()	Signal end of display list.
callobj()	glCallList(), glCallLists()	Execute display list(s).
isobj()	glIsList()	Test for display list existence.
delobj()	glDeleteLists()	Delete contiguous group of display lists.
genobj()	glGenLists()	Generate the given number of contiguous empty display lists.
—	glListBase()	Get the display list base for glCallLists().

Porting bbox2() Calls

The command **bbox2()** has no OpenGL equivalent. To port **bbox2()** calls, first create a new (OpenGL) display list that has everything that was in the corresponding IRIS GL display list except the **bbox2()** call. Then, in feedback mode, draw a rectangle the same size as the IRIS GL bounding box: if nothing comes back, the box was completely clipped and you shouldn't draw the display list.

Achieving Edited Display List Behavior

Although you can't actually edit OpenGL display lists, you can get a similar result by nesting display lists, then destroying and creating new versions of the sublists. The following OpenGL code fragment illustrates how to do this:

```
glNewList (1, GL_COMPILE);
    glIndexi (MY_RED);
glEndList ();
    glNewList (2, GL_COMPILE);
    glScalef (1.2, 1.2, 1.0);
glEndList ();

glNewList (3, GL_COMPILE);
    glCallList (1);
    glCallList (2);
glEndList ();
     .
     .
glDeleteLists (1, 2);
glNewList (1, GL_COMPILE);
    glIndexi (MY_CYAN);
glEndList ();
glNewList (2, GL_COMPILE);
    glScalef (0.5, 0.5, 1.0);
glEndList ();
```

Sample Implementation of a Display List

Example 3-5 defines three IRIS GL display lists. One display list refers to the others in its definition.

Example 3-5 IRIS GL Display Lists

```
makeobj (10);                              /* 10 object   */
   cpack (0x0000FF);
   recti (164, 33, 364, 600);              /* hollow rectangle */
closeobj ();

makeobj (20);                              /*20 object--various things*/
   cpack (0xFFFF00);
   circi(0,0,25);                          /* draw an unfilled circle */
   rectfi (100, 100, 200, 200);            /* draw filled rect */
closeobj ();

makeobj (30);                              /* 30 -- THE MAIN OBJECT */
   callobj (10);
   cpack (0xFFFFFF);
   rectfi (400, 100, 500, 300);             /* draw filled rect */
   callobj (20);
closeobj ();
                                    /* now draw by calling the lists */
callobj(30);
```

Translated to OpenGL, the code from Example 3-5 might look Example 3-6.

Example 3-6 OpenGL Display Lists

```
glNewList( 10, GL_COMPILE );
   glColor3f( 1, 0, 0 );
   glRecti( 164, 33, 364, 600 );
glEndList();

glNewList( 20, GL_COMPILE );
   glColor3f( 1, 1, 0 );               /* set color to YELLOW  */
   glPolygonMode(GL_BOTH, GL_LINE);    /* unfilled mode */
   glBegin(GL_POLYGON);           /*use polygon to approximate circle*/
      for(i=0;i<100;i++) {
         cosine = 25 * cos(i*2*PI/100.0);
         sine =   25 * sin(i*2*PI/100.0);
        glVertex2f(cosine,sine);
      }
```

```
   glEnd();

   glBegin(GL_QUADS);
      glColorf( 0, 1, 1 );                    /* set color to CYAN  */
      glVertex2i(100,100);
      glVertex2i(100,200);
      glVertex2i(200,200);
      glVertex2i(100,200);
   glEnd();
glEndList();

glNewList(30, GL_COMPILE);                     /* List #30 */
   glCallList( 10 );
     glColorf( 1, 1, 1 );                     /* set color to WHITE */
      glRecti(400, 100, 500, 300);
   glCallList( 20 );
glEndList();

                                          /* execute the display lists */
glCallList( 30 );
```

Porting defs, binds, and sets: Replacing ‘Tables’ of Stored Definitions

OpenGL doesn’t have tables of stored definitions—you cannot define lighting models, materials, textures, line styles, or patterns as separate objects as you could in IRIS GL. Thus, there are no direct equivalents to these IRIS GL calls:

- **lmdef()** and **lmbind()**
- **tevdef()** and **tevbind()**
- **texdef()** and **texbind()**
- **deflinestyle()** and **setlinestyle()**
- **defpattern()** and **setpattern()**

However, you can use display lists to mimic the def/bind behavior. (It’s often best to optimize by writing display lists that contain just a single material definition.)

For example, here is a material definition in IRIS GL:

```
float mat[] = {
    AMBIENT, .1, .1, .1,
    DIFFUSE, 0, .369, .165,
    SPECULAR, .5, .5, .5,
    SHININESS, 10,
    LMNULL
};
lmdef(DEFMATERIAL, 1, 0, mat);
lmbind(MATERIAL, 1);
```

In the following code fragment, the same material is defined in a display list, referred to by the list number in MYMATERIAL:

```
#define MYMATERIAL 10
/* you would probably use glGenLists() to get list numbers */
GLfloat   mat_amb[] = {.1, .1, .1, 1.0};
GLfloat   mat_dif[] = {0, .369, .165, 1.0};
GLfloat   mat_spec[] = { .5, .5, .5, 1.0};

glNewList( MYMATERIAL, GL_COMPILE );
    glMaterialfv( GL_FRONT, GL_AMBIENT, mat_amb);
    glMaterialfv( GL_FRONT, GL_DIFFUSE, mat_dif);
    glMaterialfv( GL_FRONT, GL_SPECULAR, mat_spec);
    glMateriali(  GL_FRONT, GL_SHININESS, 10);
glEndList();

glCallList( MYMATERIAL );
```

Porting Lighting and Materials Calls

You probably have to port lighting and materials code explicitly, because the OpenGL calls differ substantially from the IRIS GL calls. The OpenGL API has separate calls for setting lights, light models, and materials.

When porting lighting and materials calls, consider the following issues:

- OpenGL has no table of stored definitions. It has no separate **lmdef()** and **lmbind()** calls. You can use display lists to mimic the def/bind behavior. See “Porting defs, binds, and sets: Replacing ‘Tables’ of Stored Definitions” on page 67 for an example. Using display lists can have the added benefit of improving your program’s performance.

- Attenuation is now associated with each light source, rather than with the overall lighting model.
- Diffuse and specular components are separated out in OpenGL light sources.
- OpenGL light sources have an alpha component. When porting your code, it's best to set the alpha component to 1.0, indicating 100% fully opaque. That way, alpha values will be determined solely by the alpha component of your materials and the objects in your scene will look just as they did in IRIS GL.
- In IRIS GL, you could call **lmcolor()** between a call to **bgnprimitive()** and the corresponding **endprimitive()** call. In OpenGL, you can't call **glColorMaterial()** between a **glBegin()** and its corresponding **glEnd()**.

Table 3-42 lists IRIS GL lighting and materials commands and the corresponding OpenGL commands.

Table 3-42 Lighting and Materials Commands

IRIS GL Call	OpenGL Call	Meaning
lmdef(DEFLIGHT,...)	glLight()	Define a light source.
lmdef(DEFLMODEL, ...)	glLightModel()	Define a lighting model.
lmbind()	glEnable(GL_LIGHT*i*)	Enable light *i*.
lmbind()	glEnable(GL_LIGHTING)	Enable lighting.
—	glGetLight()	Get light source parameters.
lmdef(DEFMATERIAL, ...)	glMaterial()	Define a material.
lmcolor()	glColorMaterial()	Change effect of color commands while lighting is active.
—	glGetMaterial()	Get material parameters.

When the first argument for **lmbind()** is DEFMATERIAL, the equivalent command is **glMaterial()**. Table 3-43 lists the various materials parameters you can set.

Table 3-43 Material Definition Parameters

lmdef() index	glMaterial() parameter	Default	Meaning
ALPHA	GL_DIFFUSE[a]		
AMBIENT	GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color
DIFFUSE	GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	Diffuse color
SPECULAR	GL_SPECULAR[b]	(0.0, 0.0, 0.0, 1.0)	Specular color
EMISSION	GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	Emissive color
SHININESS	GL_SHININESS	0.0	Specular exponent
—	GL_AMBIENT_AND_ DIFFUSE	(see above)	Equivalent to calling glMaterial() twice with same values
COLORINDEXES	GL_COLOR_INDEXES	—	Color indices for ambient, diffuse, and specular lighting

a. The fourth value in the GL_DIFFUSE parameter specifies the alpha value.

b. In IRIS GL, if the specular exponent (i.e. SHININESS) is zero, then the specular component of the light is not added in. In OpenGL, the specular component is added in anyway.

When the first argument of **lmdef()** is DEFLMODEL, the equivalent OpenGL call is **glLightModel()**. The exception is the case when the first argument of **lmdef()** is DEFLMODEL, ATTENUATION—in this case, you have to replace **lmdef()** with several **glLight()** calls. Table 3-44 lists equivalent lighting model parameters.

Table 3-44 Lighting Model Parameters

lmdef() index	glLightModel() Parameter	Default	Meaning
AMBIENT	GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	Ambient color of scene.
ATTENUATION	—	—	See glLight().
LOCALVIEWER	GL_LIGHT_MODEL_ LOCAL_VIEWER	GL_FALSE	Viewer local (TRUE) or infinite (FALSE).
TWOSIDE	GL_LIGHT_MODEL_TWO_SIDE	GL_FALSE	Use two-sided lighting when TRUE.

When the first argument of **lmdef()** is DEFLIGHT, the equivalent OpenGL call is **glLight()**. Table 3-45 lists equivalent lighting parameters.

Table 3-45 Light Parameters

lmdef() index	glLight() Parameter	Default	Meaning
AMBIENT	GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	Ambient intensity.
	GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	Diffuse intensity.
	GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	Specular intensity.
LCOLOR			
POSITION	GL_POSITION	(0.0, 0.0, 1.0, 0.0)	Position of light.
SPOTDIRECTION	GL_SPOT_DIRECTION	(0, 0, -1)	Spot direction.
SPOTLIGHT	--		
	GL_SPOT_EXPONENT	0	Intensity distribution.
	GL_SPOT_CUTOFF	180	Maximum spread angle of light source.
DEFLMODEL, ATTENUATION, ...	GL_CONSTANT_ATTENUATION GL_LINEAR_ATTENUATION GL_QUADRATIC_ATTENUATION	(1,0,0)	Attenuation factors.

Example 3-7 is an OpenGL code fragment that demonstrates some OpenGL lighting and material calls, including two-sided lighting.

Example 3-7 OpenGL Lighting and Material Calls

```
/* Initialize lighting */
void myinit(void)
{
    GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
/*      light_position is NOT default value     */
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
```

```
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    glFrontFace (GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}

void display(void)
{
    GLdouble eqn[4] = {1.0, 0.0, -1.0, 1.0};
    GLfloat mat_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
    GLfloat back_diffuse[] = { 0.8, 0.2, 0.8, 1.0 };

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix ();
    glClipPlane (GL_CLIP_PLANE0, eqn);  /*  slice objects   */
    glEnable (GL_CLIP_PLANE0);

    glPushMatrix ();
    glTranslatef (0.0, 2.0, 0.0);
    auxSolidTeapot(1.0);        /*  one-sided lighting  */
    glPopMatrix ();

        /*  two-sided lighting, but same material       */
    glLightModelf (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
    glMaterialfv (GL_FRONT_AND_BACK, GL_DIFFUSE,
                  mat_diffuse);
    glPushMatrix ();
    glTranslatef (0.0, 0.0, 0.0);
    auxSolidTeapot(1.0);
    glPopMatrix ();

        /*  two-sided lighting, two different materials */
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv (GL_BACK, GL_DIFFUSE, back_diffuse);
    glPushMatrix ();
    glTranslatef (0.0, -2.0, 0.0);
```

```
    auxSolidTeapot(1.0);
    glPopMatrix ();

    glLightModelf (GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
    glDisable (GL_CLIP_PLANE0);
    glPopMatrix ();
    glFlush();
}
```

Porting Texture Calls

When porting texture calls, consider these issues:

- At times, a single IRIS GL texturing call has to be replaced with two or more OpenGL calls. For those cases, edit the *toogl* output to use more variables than it did before or restructure the program.
- Use **glEnable()** and **glDisable()** to turn texturing capabilities on and off. See the reference page for details.
- OpenGL doesn't automatically generate mipmaps for you—if you're using mipmaps, call **gluBuild2DMipmaps()** first.
- Texture size in OpenGL is more strictly regulated than in IRIS GL. An OpenGL texture must be

  $2^n + 2b$

  where n is an integer and b is

  – 0, if there's no border

  – 1, if there's a border pixel (textures in OpenGL can have 1 pixel borders)
- OpenGL 1.0 keeps no tables of textures, just a single 1D texture and a single 2D texture. If you want to reuse your textures, put them in a display list, as described in "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67.
- In OpenGL 1.1, you can use named textures. Use the functions **glBindTexture()**, **glGenTexture()**, and **glDeleteTextures()** to work with named texture object. You can also call **glPrioritizeTextures()** to have certain textures preferentially be assigned to texture memory, and **glAreTexturesResident()** to determine whether a named texture is currently in texture memory.
- In OpenGL 1.1, you can use the subtexture mechanism for more efficient texture loading.

- OpenGL 1.1 offers the proxy texture mechanism to let you test whether a texture will fit into texture memory on a certain system.

Table 3-46 lists the general OpenGL equivalents to IRIS GL texture calls.

Table 3-46 Texture Commands

IRIS GL Call	OpenGL Call	Meaning
texdef2d()	glTexImage2D() glTexParameter() gluBuild2DMipmaps()	Specify a 2D texture image.
texbind()	glTexParameter() glTexImage2D() gluBuild2DMipmaps()	Select a texture function.
tevdef()	glTexEnv()	Define a texture mapping environment.
tevbind()	glTexEnv()	Select a texture environment.
—	glTexImage1D()	
t2*(), t3*(), t4*()	glTexCoord*()	Set the current texture coordinates.
texgen()	glTexGen()	Control generation of texture coordinates.
—	glGetTexParameter()	—
—	gluBuild1DMipmaps()	—
—	gluBuild2DMipmaps()	—
—	gluScaleImage()	Scale an image to arbitrary size.

The *OpenGL Programming Guide* describes in detail how textures work in OpenGL. Here are a few general tips:

- Remember to call **gluBuild2DMipmaps()** or **gluBuild1DMipmaps()** before trying to use mipmaps.
- Use **glTexParameter()** to specify wrapping and filters.
- Use **glTexEnv()** to set up texturing environment.
- Use **glTexImage2D()** or **glTexImage1D()** to load each image.

- Use **glEnable()** and **glDisable()** to turn texturing capabilities on and off.

See the reference page for each call for detailed information.

Translating tevdef()

Here's an example of an IRIS GL texture environment definition that specifies the TV_DECAL texture environment option:

```
float tevprops[] = {TV_DECAL, TV_NULL};
tevdef(1, 0, tevprops);
```

Here's how you could translate that code to OpenGL:

```
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

Table 3-47 lists the IRIS GL texture environment options and their OpenGL equivalents.

Table 3-47 Texture Environment Options

IRIS GL Option	OpenGL Option
TV_MODULATE	GL_MODULATE
TV_DECAL	GL_DECAL
TV_BLEND	GL_BLEND
TV_COLOR	GL_TEXTURE_ENV_COLOR
TV_ALPHA	no direct OpenGL equivalent
TV_COMPONENT_SELECT	no direct OpenGL equivalent

For more detailed information on how to use these options, see the glTexEnv reference page.

Translating texdef()

Here's an example of an IRIS GL texture definition:

```
float texprops[] = { TX_MINFILTER, TX_POINT,
                     TX_MAGFILTER, TX_POINT,
                     TX_WRAP_S, TX_REPEAT,
                     TX_WRAP_T, TX_REPEAT,
                     TX_NULL };
texdef2d(1, 1, 6, 6, granite_texture, 7, texprops)
```

In the above code example, **texdef()** specifies the TX_POINT filter as both the magnification and the minification filter, and TX_REPEAT as the wrapping behavior. It also specifies the texture image, in this case an image called *granite_texture*.

In OpenGL, the image specification is handled by the **glTexImage*()** functions and property-setting is handled by **glTexParameter()**. To translate to OpenGL, you'd replace a **texdef()** call with a call to a **glTexImage*()** routine and one or more calls to **glTexParameter()**.

Here's an example of one way you could translate the IRIS GL code fragment above:

```
GLfloat nearest [] = {GL_NEAREST};
GLfloat repeat [] = {GL_REPEAT};
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                  nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                  nearest);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S,
                  repeat);
glTexParameterfv( GL_TEXTURE_1D, GL_TEXTURE_WRAP_T,
                  repeat);
glTexImage1D(GL_TEXTURE_1D, 0, 1, 6, 0, GL_RGB,
                  GL_UNSIGNED_SHORT,granite_tex);
```

Table 3-48 lists the IRIS GL texture parameters with their OpenGL equivalents. For more detailed information on OpenGL texture parameters, see the glTexParameter reference page.

Table 3-48 IRIS GL and OpenGL Texture Parameters

texdef(... np, ...) Option	glTexParameter() Parameter Name
TX_MINFILTER	GL_TEXTURE_MIN_FILTER
TX_MAGFILTER	GL_TEXTURE_MAG_FILTER
TX_WRAP, TX_WRAP_S	GL_TEXTURE_WRAP_S
TX_WRAP, TX_WRAP_T	GL_TEXTURE_WRAP_T
—	GL_TEXTURE_BORDER_COLOR

Table 3-49 lists the possible values of the IRIS GL texture parameters along with their OpenGL equivalents. If you used special values available only on systems with RealityEngine™ graphics, you may have to wait for RealityEngine extensions to OpenGL before you can translate these values exactly (see "Porting RealityEngine Graphics Features" for further discussion). For more information on possible values of OpenGL texture parameters, see the glTexParameter reference page.

Table 3-49 Values for IRIS GL and OpenGL Texture Parameters

IRIS GL	OpenGL
TX_POINT	GL_NEAREST
TX_BILINEAR	GL_LINEAR
TX_MIPMAP_POINT	GL_NEAREST_MIPMAP_NEAREST
TX_MIPMAP_BILINEAR	GL_LINEAR_MIPMAP_NEAREST
TX_MIPMAP_LINEAR	GL_NEAREST_MIPMAP_LINEAR
TX_TRILINEAR	GL_LINEAR_MIPMAP_LINEAR

Translating texgen()

The functionality of **texgen()** is replaced by **glTexGen()** almost entirely, though you have to call **glEnable()** and **glDisable()** to turn coordinate generation on and off. Table 3-50 lists the equivalents for texture coordinate names.

Table 3-50 Texture Coordinate Names

IRIS GL Texture Coordinate	OpenGL Texture Coordinate	glEnable() Argument
TX_S	GL_S	GL_TEXTURE_GEN_S
TX_T	GL_T	GL_TEXTURE_GEN_T
TX_R	GL_R	GL_TEXTURE_GEN_R
TX_Q	GL_Q	GL_TEXTURE_GEN_Q

Table 3-51 lists texture generation mode and plane names.

Table 3-51 Texture Generation Modes and Planes

IRIS GL Texture Mode	OpenGL Texture Mode	Corresponding Plane Name
TG_LINEAR	GL_OBJECT_LINEAR	GL_OBJECT_PLANE
TG_CONTOUR	GL_EYE_LINEAR	GL_EYE_PLANE
TG_SPHEREMAP	GL_SPHERE_MAP	—

With IRIS GL, you call **texgen()** twice: once to simultaneously set the mode and a plane equation, and once more to enable texture coordinate generation. In OpenGL, you make three calls: two to **glTexGen()** (once to set the mode, and again to set the plane equation), and one to **glEnable()**. For example, if you called **texgen()** like this:

```
texgen(TX_S, TG_LINEAR, planeParams);
texgen(TX_S, TG_ON, NULL);
```

the equivalent OpenGL code is:

```
glTexGen(GL_S, GL_TEXTURE_GEN_MODE, modeName);
glTexGen(GL_S, GL_OBJECT_PLANE, planeParams);
glEnable(GL_TEXTURE_GEN_S);
```

Porting Picking Calls

All the IRIS GL picking calls have OpenGL equivalents, with the exception of **clearhitcode()**. Table 3-52 lists the IRIS GL picking calls and their OpenGL counterparts.

Table 3-52 Calls for Picking

IRIS GL Call	OpenGL Call	Notes
clearhitcode()	not supported	Clears global variable, hitcode.
pick(), select()	glRenderMode(GL_SELECT)	Switch to selection/picking mode.
endpick(), endselect()	glRenderMode(GL_RENDER)	Switch back to rendering mode.
picksize()	gluPickMatrix()	
—	glSelectBuffer()	Set the return array.
initnames()	glInitNames()	—
pushname(), popname()	glPushName(), glPopName()	—
loadname()	glLoadName()	—

For more information on picking, refer to the gluPickMatrix reference page and the *OpenGL Programming Guide*. You can find a complete example and additional explanation in "OpenGL Programming for the X Window System," page 438-441. See "GLX and GLUT Documentation" on page xvi.

Porting Feedback Calls

Feedback under IRIS GL differed from machine to machine. OpenGL standardizes feedback, so you can now rely on consistent feedback from machine to machine, and from implementation to implementation. Table 3-53 lists IRIS GL and OpenGL feedback calls.

Table 3-53 Feedback Calls

IRIS GL Call	OpenGL Call	Notes
feedback()	glRenderMode(GL_FEEDBACK)	Switch to feedback mode.
endfeedback()	glRenderMode(GL_RENDER)	Switch back to rendering mode.
—	glFeedbackBuffer()	—
passthrough()	glPassThrough()	Place a token marker in the feedback buffer.

For more information, see the reference pages or the *OpenGL Programming Guide*.

Example 3-8 demonstrates OpenGL feedback.

Example 3-8 Feedback in OpenGL

```
/*
 * feedback.c
 * This program demonstrates use of OpenGL feedback.  First,
 * a lighting environment is set up and a few lines are drawn.
 * Then feedback mode is entered, and the same lines are
 * drawn.  The results in the feedback buffer are printed.
 */
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>

/*  Initialize lighting.
 */
void init(void)
{
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
}

/* Draw a few lines and two points, one of which will
```

```
 * be clipped.  If in feedback mode, a passthrough token
 * is issued between the each primitive.
 */
void drawGeometry(GLenum mode)
{
   glBegin(GL_LINE_STRIP);
   glNormal3f(0.0, 0.0, 1.0);
   glVertex3f(30.0, 30.0, 0.0);
   glVertex3f(50.0, 60.0, 0.0);
   glVertex3f(70.0, 40.0, 0.0);
   glEnd();
   if (mode == GL_FEEDBACK)
      glPassThrough(1.0);
   glBegin(GL_POINTS);
   glVertex3f(-100.0, -100.0, -100.0);  /*  will be clipped  */
   glEnd();
   if (mode == GL_FEEDBACK)
      glPassThrough(2.0);
   glBegin(GL_POINTS);
   glNormal3f(0.0, 0.0, 1.0);
   glVertex3f(50.0, 50.0, 0.0);
   glEnd();
}

/* Write contents of one vertex to stdout.
*/
void print3DcolorVertex(GLint size, GLint *count, GLfloat *buffer)
{
   int i;

   printf("  ");
   for (i = 0; i < 7; i++) {
      printf("%4.2f ", buffer[size-(*count)]);
      *count = *count - 1;
   }
   printf ("\n");
}

/*  Write contents of entire buffer. (Parse tokens!)
*/
void printBuffer(GLint size, GLfloat *buffer)
{
   GLint count;
   GLfloat token;
```

```
   count = size;
   while (count) {
      token = buffer[size-count];
      count--;
      if (token == GL_PASS_THROUGH_TOKEN) {
         printf("GL_PASS_THROUGH_TOKEN\n");
         printf("  %4.2f\n", buffer[size-count]);
         count--;
      } else if (token == GL_POINT_TOKEN) {
         printf("GL_POINT_TOKEN\n");
         print3DcolorVertex (size, &count, buffer);
      } else if (token == GL_LINE_TOKEN) {
         printf("GL_LINE_TOKEN\n");
         print3DcolorVertex(size, &count, buffer);
         print3DcolorVertex(size, &count, buffer);
      } else if (token == GL_LINE_RESET_TOKEN) {
         printf("GL_LINE_RESET_TOKEN\n");
         print3DcolorVertex(size, &count, buffer);
         print3DcolorVertex(size, &count, buffer);
      }
   }
}

void display(void)
{
   GLfloat feedBuffer[1024];
   GLint size;

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0.0, 100.0, 0.0, 100.0, 0.0, 1.0);

   glClearColor(0.0, 0.0, 0.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   drawGeometry(GL_RENDER);

   glFeedbackBuffer(1024, GL_3D_COLOR, feedBuffer);
   glRenderMode(GL_FEEDBACK);
   drawGeometry(GL_FEEDBACK);

   size = glRenderMode(GL_RENDER);
   printBuffer(size, feedBuffer);
}

/* ARGSUSED1 */
```

```
void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(100, 100);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Porting RealityEngine Graphics Features

Some IRIS GL features that were available only on systems with RealityEngine graphics are unavailable in OpenGL; though several of them are supported either by OpenGL 1.1 or by an extension.

Table 3-54 lists the IRIS GL RealityEngine calls and their OpenGL counterparts.

Table 3-54 RealityEngine Calls

IRIS GL Call	OpenGL Call	Notes
blendcolor()	glBlendColorEXT()	Specify a color to blend.
convolve()	glConvolutionFilter2DEXT(), glSeparableFilter2DEXT(), glConvolutionParameterEXT(), glPixelTransfer()	Convolve an input image with a kernel image.

Table 3-54 (continued) RealityEngine Calls

IRIS GL Call	OpenGL Call	Notes
displacepolygon()	glPolygonOffsetEXT() (OpenGL 1.0) glPolygonOffset() (OpenGL 1.1)	Specify *z* displacement for rendered polygons.
fbsubtexload()	Not supported	Load part or all of a texture.
gethgram()	glGetHistogramEXT()	Get histogram data.
getminmax()	glGetMinmaxEXT()	Get minimum and maximum graphics values.
hgram()	glHistogramEXT(), glResetHistogramEXT()	Compute histogram of pixel-transfer information.
ilbuffer()	Not supported	Allocate space for temporary image-processing results.
ildraw()	Not supported	Select an ilbuffer to draw into.
istexloaded()	glAreTexturesResidentEXT (OpenGL 1.0) glAreTexturesResident (OpenGL 1.1)	Find out whether a given texture is resident in texture memory.
leftbuffer()	glDrawBuffer(GL_LEFT)	Enable left-buffer drawing.
minmax()	glMinmaxEXT()	Compute minimum and maximum pixel values.
monobuffer()	Superseded by selection of an appropriate GLX visual	Select monoscopic viewing.
msalpha()	glEnable(GL_SAMPLE_ALPHA_TO_MASK_SGIS), glEnable(GL_SAMPLE_ALPHA_TO_ONE_SGIS)	Specify treatment of alpha values during multisampling.
msmask()	glSampleMaskSGIS()	Specify a multisample mask.
mspattern()	glSamplePatternSGIS()	Specify a sample pattern for multisampling.
mssize()	glXChooseVisual() with attribute GLX_SAMPLE_BUFFERS_SGIS	Configure multisample buffer.
multisample()	glEnable(GL_MULTISAMPLE_SGIS)	Enable or disable multisampling.

Table 3-54 (continued) RealityEngine Calls

IRIS GL Call	OpenGL Call	Notes
pixelmap()	glPixelMap()	Define pixel transfer lookup tables.
pixeltransfer()	glPixelTransfer()	Set transfer modes.
readcomponent()	glReadPixels() gives partial support; some readcomponent() features aren't yet supported	Choose a component source.
rightbuffer()	glDrawBuffer(GL_RIGHT)	Enable drawing in right buffer.
stereobuffer()	Superseded by selection of an appropriate GLX visual.	Select stereoscopic viewing.
subtexload()	OpenGL 1.1 function glTexSubImage2D() gives partial support (the *flags* parameter to subtexload() isn't supported)	Load part or all of a texture.
texdef3d()	glTexImage3DEXT()	Convert 3D image into a texture.
tlutbind()	Not supported	Select a texture lookup table.
tlutdef()	Not supported	Define a texture lookup table.
zbsize()	Superseded by selection of an appropriate GLX visual	Specify number of bitplanes to use for the depth buffer.

Some RealityEngine features (mostly involving texturing) don't correspond to specific IRIS GL functions, and thus don't fit nicely into Table 3-54. Some such features are supported by extensions to OpenGL; you should check at runtime to see if the relevant extension is supported by calling `glGetString(GL_EXTENSIONS)` (see the glGetString reference page for more information). Some other non-function-specific IRIS GL RealityEngine features aren't supported at all.

Each of the following features is supported on a given machine if the corresponding OpenGL extension is supported. ("OpenGL Extensions" on page 87 points you to additional information):

- The internal texture storage format (TX_INTERNAL_FORMAT in IRIS GL) is supported by the texture extension (GL_EXT_texture). OpenGL without extensions

supports a superset of the formats previously specified by TX_EXTERNAL_FORMAT; see the glTexImage2D reference page for more information.

- Sharpen texture is supported by the GL_SGIS_sharpen_texture extension. This was done in IRIS GL by passing TX_SHARPEN to **texdef()**.
- Detail texture is supported by the GL_SGIS_detail_texture extension. This was done in IRIS GL by using the tokens TX_DETAIL, TX_ADD_DETAIL, and TX_MODULATE_DETAIL in **texdef()** calls.
- The detail texture and sharpen texture extension both support control points (pairs of level-of-detail and scale values) to control the rate at which the relevant filters are applied (see TX_CONTROL_CLAMP and TX_CONTROL_POINT in the **texdef()** reference page). However, unlike IRIS GL, OpenGL uses a separate set of control points for each of the two filters.
- The IRIS GL ABGR pixel format is supported by the GL_EXT_abgr extension.
- Texture and texture environment definition and binding (formerly done by using **texdef()**, **texbind()**, **tevdef()**, and **tevbind()**) are currently handled in OpenGL by creating a display list containing a **glTexImage2D()** call. (No OpenGL extension is required.)
- The texture object extension supports named textures and allows you to prioritize textures using **glPrioritizeTexturesEXT()**. This functionality is also part of OpenGL 1.1.

These features are not supported in OpenGL or its extensions:

- Automatic mipmap generation is supported in the GLU library by **gluBuild2DMipmaps()**, but you can't change the default filtering used to generate mipmap levels (see TX_MIPMAP_FILTER_KERNEL in the texdef reference page).
- Bicubic texture filtering (see the descriptions of TX_BICUBIC and TX_BICUBIC_FILTER in the texdef reference page).
- shadows (see the descriptions of TX_BILINEAR_LEQUAL and TX_BILINEAR_GEQUAL in the **texdef()** reference page, and of TV_ALPHA in the **tevdef()** reference page).
- Component selection (see TV_COMPONENT_SELECT in the **tevdef()** reference page).
- Texture definition from a live video stream (available in IRIS GL using the *flags* argument to **subtexload()**).

On some platforms, the video source extension, SGIX_video_source, lets you source pixel data from a video stream to the OpenGL renderer. The video source extension is available only for system configurations that have direct hardware paths from the video hardware to the graphics accelerator. On other systems, you need to transfer video data to host memory and then call **glDrawPixels()** or **glTex{Sub}Image()** to transfer data to the framebuffer or to texture memory.

- Fast texture definition, as performed in IRIS GL with TX_FAST_DEFINE.
- Quadlinear mipmap filtering (see TX_MIPMAP_QUADLINEAR in the texdef reference page).
- Specifying separate alpha and non-alpha functions for texture magnification filtering (see TX_MAGFILTER_COLOR and TX_MAGFILTER_ALPHA in the texdef reference page).

OpenGL Extensions

For information on extensions to OpenGL, see the glintro and glxintro reference pages, as well as the reference pages for individual functions. (For a partial list of extension-related functions, see "Porting RealityEngine Graphics Features.")

The manual *OpenGL on Silicon Graphics Systems* discusses each extension and explains how to use it.

Chapter 4

OpenGL in the X Window System

This chapter provides some information about OpenGL programming in the X environment. The chapter focuses on information relevant to translating IRIS GL programs into programs using OpenGL and X—it doesn't provide a tutorial on Xt and IRIS IM.

This chapter discusses the following topics:

- "X Window System Background" on page 90 provides some basic information about the X Window system and also briefly discusses porting issue, naming conventions, and the two porting options (widget or Xlib).
- "Advice for OpenGL Programs using the X Window System" on page 92 discusses window depth, display mode, and X colormaps.
- "Fonts and Strings" on page 92 explores using fonts in an OpenGL program.
- "Using Xt and a Widget Set" on page 94 discusses using a widget set in more detail.
- "Using Xlib and GLX Commands" on page 103 discusses using straight OpenGL in more detail.

Several documents can help you with more detailed information (see "Where to Get More Information" on page xiv):

- For a detailed discussion of all issues involving OpenGL and X, see "OpenGL Programming for the X Window System" discussed under "GLX and GLUT Documentation" on page xvi.
- For more information on the relevant features of Xt and IRIS IM, consult the OSF/Motif series, and Digital's *X Window System Toolkit: The Complete Programmer's Guide and Specification*, or O'Reilly's Volumes 4 and 5 on X Toolkit Intrinsics.
- For information on OpenGL in the X Window System within a Silicon Graphics environment, see *OpenGL on Silicon Graphics Systems*.

X Window System Background

An X program can create one or more subwindows that use OpenGL for rendering. Such a program allows full access to the capabilities of X by completely removing OpenGL from any feature governed by the X server. You have direct control of all the areas governed by the X server: the event handling, window control, and menus. You also use X to handle color maps and fonts.

You can find examples of programs that use either OpenGL or IRIS GL in the *usr/share* directory and through the SGI home page.

In IRIS GL, you could use IRIS GL event and window management routines, such as **winopen()** or **qread()**—which would access the X Window System for you. In OpenGL, you can use the GLUT library for all basic windowing and event handling routines. If the GLUT library isn't sufficient for your purposes, you can modify your IRIS GL code to be an OpenGL program using GLX.

Function Naming Conventions

IRIS GL can draw in X11 windows with routines such as **GLXgetconfig()**, **GLXlink()**, **GLXunlink()**, and **GLXwinset()**. These functions don't have exact equivalents in OpenGL; see Appendix A for approximate equivalents.

The naming conventions for X-related functions may be confusing, as they depend largely on capitalization to differentiate between groups of functions:

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 **glX*()** routines are, collectively referred to as "GLX." Note, too, that **GLXgetconfig()** (an IRIS GL mixed-model routine) isn't at all the same function as **glXGetConfig()** (an OpenGL GLX routine). The command

```
IRIS% man glxgetconfig
```

on a system with both IRIS GL and OpenGL lists both reference pages, one following the other.

Two Choices for Using OpenGL and X

When integrating your OpenGL program with the X Window System, you have two choices:

- Use the Xt toolkit and a widget set, such as IRIS IM (see "Using Xt and a Widget Set")
- Write your program in Xlib and OpenGL using special GLX commands (see "Using Xlib").

The first method, using Xt and a widget set, is easier and is commonly used by developers. It's recommended particularly for programmers with little or no previous experience with X.

Note: The manual *OpenGL on Silicon Graphics Systems* explores both approaches in some detail.

Whichever method you choose, you'll find more information on programming with Xlib and Xt in the X Window System series from O'Reilly & Associates. The material in this chapter is intended as a supplement to the O'Reilly guides, detailing X development features available on Silicon Graphics workstations.

Using Xt and a Widget Set

Silicon Graphics provides a widget library that simplifies programming with Xt. "Using Xt and a Widget Set" on page 94 explains how to convert your IRIS GL program to an OpenGL program using Xt, the IRIS Widget Library, and the GL widget, GLwDrawingArea (Silicon Graphics also provides an IRIS IM—Motif—version of GLwDrawingArea, called GLwMDrawingArea.)

Using Xlib

If you prefer to use Xlib without using Xt (in effect working at a lower level), refer to the recommended references on X programming, and use the GLX routines described in the *OpenGL Reference Manual* (start with the glXIntro reference page). "Using Xlib and GLX Commands" on page 103 provides more information and contains some code examples. Several complete programs using this method are included in Appendix F, "Example Mixed-Model Programs With Xlib," along with IRIS GL versions of the same programs.

Advice for OpenGL Programs using the X Window System

This section briefly discusses two important issues:

- "Dealing With Window Depth and Display Mode"
- "Installing Color Maps"

Dealing With Window Depth and Display Mode

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

Installing Color Maps

It's a good idea to call **XSetWMColormapWindows()**; this ensures that its color maps are installed. If you don't call **XSetWMColormapWindows()**, the default X color map is used. Even if your program uses RGB mode, you should still call **XSetWMColormapWindows()** because some hardware (such as IRIS Indigo) simulates RGB with a color map.

Fonts and Strings

OpenGL contains no equivalents for the IRIS GL text-handling calls and Font Manager calls. To obtain full text- and font-handling facilities, call **glXUseXFont()** with display lists to get some text-display capabilities. You can also use the GLUT font rendering calls for some more limited text- and font-handling facilities.

This section gives you an example; to use display lists to do X bitmap fonts, your program should do the following:

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

3. Put the bitmap for one character into each display list, in the order the characters appear in the font.
4. To print out 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()**.

The following code fragment gives you an example, using Adobe Times Medium to print out the string "The quick brown fox jumps over a lazy dog." It also prints out the entire character set, from ASCII 32 to 127.

Example 4-1 OpenGL Character Rendering

```
#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 ();
}
```

Note: You can also use the OpenGL character renderer (glc) to render characters. See the glcintro reference page for more information.

Using Xt and a Widget Set

In general, you can bypass many of the complexities of X by using the Xt toolkit and a widget set such as IRIS IM.

When mixing OpenGL with Xt, IRIS IM, or Athena widgets, you can use the Silicon Graphics GLwDrawingArea widget, which simplifies programming with IRIS IM or any other widget set. The GLwDrawingArea widget is also compatible with User Interface Language (UIL). This section explains how to use the GLwMDrawingArea widget for embedding GL in an Xt or IRIS IM program. It discusses these topics:

What You Need to Know About Xt and IRIS IM

The examples shown in this chapter use Xt and IRIS IM. Although knowing Xt and IRIS IM isn't required to read this chapter, understanding the details of the examples does require some Xt and IRIS IM knowledge. This chapter points out a few areas of the Xt and IRIS IM toolkits relevant for OpenGL programmers; it doesn't provide a tutorial on Xt and IRIS IM.

For more information on the relevant features of Xt and IRIS IM, consult the OSF/Motif series, and Digital's *X Window System Toolkit: The Complete Programmer's Guide and Specification*, or O'Reilly's Volumes 4 and 5 on the X Toolkit Intrinsics, or refer to the *OpenGL on Silicon Graphics Systems* manual.

About Xt

Xt, also known as the X Toolkit Intrinsics, is a C library that provides routines for creating and using user interface components called *widgets*. It's usually easier to convert your IRIS GL program using Xt than it is to use the low-level Xlib programming library.

Since Xt doesn't dictate the "look and feel" of the GUI, you must use it in conjunction with a widget set (a library of pre-built widgets), such as the Athena widget set or IRIS IM.

About IRIS IM

IRIS IM is the Silicon Graphics port of OSF/Motif. Motif is an extensible widget set of user interface objects, such as buttons, scroll bars, menu systems, and dialog boxes. These widgets are accessible via a library of C routines. These widgets are supported by Xt. Ultimately, the X Window System is the foundation for both the Motif and Athena widget sets.

Motif is also a style guide, which describes the "look and feel" of a Motif compliant user interface.

IRIS IM and Other Widget Sets

This section refers frequently to IRIS IM because it is commonly used with OpenGL programs; however, unless otherwise specified, you can use the features discussed here with other widget sets, such as the Athena widget set. The features discussed in this chapter exist either within the widget itself or are based on the X toolkit. You therefore have a choice:

- Use the generic GLwDrawingArea widget.
- Use the IRIS IM (Motif) widget GLw**M**DrawingArea.

Combining OpenGL and Motif is made easier by a specially supplied OpenGL drawing area widget, GLwDrawingArea. Use the GLwDrawingArea widget when integrating your OpenGL program with Xt. The GLwDrawingArea widget sets up a configuration for GL drawing and provides resources and callbacks that are useful to the OpenGL programmer. The GLwDrawingArea widget also provides support for overlays.

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

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

- GLwMDrawingArea is a subclass of the IRIS IM XmPrimitive widget, rather than being a subclass of the Xt Core widget and, therefore, has various defaults such as background color.
- GLwMDrawingArea understands IRIS IM traversal, although traversal is turned off by default.
- You can create GLwMDrawingArea directly through Xt or use an IRIS IM creation function, **GlxCreateMDrawingArea()**.

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

Converting Your IRIS GL Program

This section discusses the actual conversion process:

- "Finding Areas for Porting" explains how your can determine which parts of your IRIS GL program you need to replace with X.
- "Using the OpenGL Widget" provides an example program of OpenGL inside the OpenGL widget.

Finding Areas for Porting

When porting to OpenGL, you have to replace any IRIS GL windowing and event handling code. One way to do this is to run *toogl* and then search through the output for the *toogl* warnings marked "OGLXXX." It should be reasonably straightforward to determine which warnings relate to X.

Using the OpenGL Widget

This section shows a simple example of a program that uses the IRIS IM version of the OpenGL widget and explains how the code works.

The generic version of the widget can be used in the same way. To compile this example, use this command line:

```
% cc -O -o mixed mixed.c -lXm -lGL -lGLw -lGLU
```

When the OpenGL widget is initially opened, its visual must be set. In other words, you must first declare the display mode of the visual: single or double buffer, color index or RGBA mode. You may also specify how many bits will be used by the components of the frame buffer: for example, depth, stencil, and accumulation bits.

In the program shown in Example 4-2, the function **init_window()**, which is registered with the GlxNginitCallback callback, calls **glXCreateContext()** to set the visual of the OpenGL widget. In this case, the resources for the widget are set to support RGBA and double buffer mode. (See the *fallback_resources[]* array in the **main()** procedure.)

Example 4-2 OpenGL Program Using IRIS IM OpenGL Widget

```
/* mixed.c
 */

#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <X11/keysym.h>
#include <X11/StringDefs.h>
#include "GL/GLwMDrawA.h"
```

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

static void input(Widget, XtPointer, XtPointer);
static void draw_scene_callback (Widget, XtPointer,
                                 XtPointer);
static void do_resize(Widget, XtPointer, XtPointer);
static void init_window(Widget, XtPointer, XtPointer);

static GLXContext glx_context;

void main(int argc, char** argv)
{
    Arg args[20];
    int n;
    Widget glw, toplevel, form;
    static XtAppContext app_context;
    static String fallback_resources[] = {
        "*glwidget*width: 300",
        "*glwidget*height: 300",
        "*glwidget*rgba: TRUE",
        "*glwidget*doublebuffer: TRUE",
        "*glwidget*allocateBackground: TRUE",
        NULL
    };

    toplevel = XtAppInitialize(&app_context, "Mixed", NULL,
                               0, &argc, argv,
                               fallback_resources, NULL, 0);
    n = 0;
    form = XmCreateForm(toplevel, "form", args, n);
    XtManageChild(form);

    n = 0;
    XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM);
    n++;
    XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM);
    n++;
    XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM);
    n++;
    XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM);
    n++;
```

```
    glw = GLwCreateMDrawingArea(form, "glwidget", args, n);
    XtManageChild (glw);
    XtAddCallback(glw, GLwNexposeCallback,
                  draw_scene_callback, (XtPointer) NULL);
    XtAddCallback(glw, GLwNresizeCallback, do_resize,
                  (XtPointer) NULL);
    XtAddCallback(glw, GLwNginitCallback, init_window,
                  (XtPointer) NULL);
    XtAddCallback(glw, GLwNinputCallback, input,
                  (XtPointer) NULL);

    XtRealizeWidget(toplevel);
    XtAppMainLoop(app_context);
}

static int rotation = 0;

void spin (void)
{
    rotation = (rotation + 5) % 360;
}

static void draw_scene (Widget w)
{
    GLUquadricObj *quadObj;

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glPushMatrix();
    glTranslatef (0.0, 0.0, -5.0);
    glRotatef ((GLfloat) rotation, 1.0, 0.0, 0.0);

    glPushMatrix ();
    glRotatef (90.0, 1.0, 0.0, 0.0);
    glTranslatef (0.0, 0.0, -1.0);
    quadObj = gluNewQuadric ();
    gluQuadricDrawStyle (quadObj, GLU_LINE);
    gluCylinder (quadObj, 1.0, 1.0, 2.0, 12, 2);
    glPopMatrix ();

    glPopMatrix();
    glFlush();
    glXSwapBuffers (XtDisplay(w), XtWindow(w));
}
```

```
/* Process all Input callbacks*/
static void input(Widget w, XtPointer client_data,
                  XtPointer call)
{
    char buffer[1];
    KeySym keysym;
    GLwDrawingAreaCallbackStruct *call_data;

    call_data = (GLwDrawingAreaCallbackStruct *) call;

    switch(call_data->event->type)
    {
    case KeyRelease:
         /* It is necessary to convert the keycode to a
          * keysym before it is possible to check if it is
          * an escape.
          */
         if (XLookupString( (XKeyEvent *) call_data->event,
                            buffer, 1, &keysym,
                            (XComposeStatus *) NULL ) == 1
             && keysym == (KeySym) XK_Escape)
             exit(0);
    break;

    case ButtonPress:
        switch (call_data->event->xbutton.button)
        {
        case Button1:
            spin();
            draw_scene(w);
        break;
        }
    break;

    default:
    break;
    }
}

static void draw_scene_callback(Widget w, XtPointer client_data,
XtPointer call)
{
    static char firstTime = 0x1;
    GLwDrawingAreaCallbackStruct *call_data;
```

```
    call_data = (GLwDrawingAreaCallbackStruct *) call;
    GLwDrawingAreaMakeCurrent(w, glx_context);

    if (firstTime) {
        glViewport(0, 0, call_data->width,call_data->height);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(65.0, (float) call_data->width /
                       (float)call_data->height, 1.0, 20.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        firstTime = 0;
    }
    draw_scene (w);
}

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

    call_data = (GLwDrawingAreaCallbackStruct *) call;

    GLwDrawingAreaMakeCurrent(w, glx_context);
    glViewport(0, 0, call_data->width, call_data->height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(65.0, (GLfloat) call_data->width /
                   (GLfloat)call_data->height, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

static void init_window(Widget w, XtPointer client_data,
                        XtPointer call_data)
{
    Arg args[1];
    XVisualInfo *vi;
    GLUquadricObj *quadObj;

    XtSetArg(args[0], GLwNvisualInfo, &vi);
    XtGetValues(w, args, 1);
    glx_context = glXCreateContext(XtDisplay(w), vi, 0,
                                   GL_FALSE);
}
```

It's a good idea to always call **GlxDrawingAreaMakeCurrent()** to set the current widget. In Example 4-2, **GlxDrawingAreaMakeCurrent()** is called from the callback functions.

Example 4-2 draws a wire frame cylinder using OpenGL. The GlxNinputCallback calls **input()**, which handles mouse and keyboard input. Pressing the Escape key causes the program to exit. Pressing Button1 (usually the left mouse button) calls **spin()**, which changes the rotation of the cylinder. Then the scene is completely redrawn.

The *mixed.c* program has absolutely basic placement of widgets. The OpenGL drawing area widget is attached to all sides of its parent, an IRIS IM XmForm widget. This is a minimal arrangement—you can add additional IRIS IM widgets for a more sophisticated interface.

You might also want to add a WorkProc (or idle) function, which executes when no other events are occurring. A WorkProc is useful for rendering continuous motion, which doesn't require steady input events; for example, an animation. Appendix E, "Example Program Using Xt and a WorkProc," contains an example program using Xt and a WorkProc.

Background Reading

The most complete information about OpenGL and X can be found in

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

For more information on mixed-model programming in general, you can refer to the *OpenGL Reference Manual*, which contains reference pages for the OpenGL GLX command, as well as an introductory reference page, glXIntro.

For more detailed information on programming with Xt, see Volume IV of the X Window System Series, *X Toolkit Intrinsics Programming Manual*, by Adrian Nye and Tim O'Reilly, published by O'Reilly & Associates, Inc. (If you're using IRIS IM, you'll probably want the Motif version of Volume IV.)

For more information on IRIS IM, refer to documentation on Motif, such as the OSF/Motif Series published by Prentice Hall.

Using Xlib and GLX Commands

Using Xlib and GLX can be more complex than using Xt and a widget set, and Silicon Graphics doesn't recommend it unless you're already familiar with Xlib programming. This section provides an overview of the necessary steps for using Xlib and GLX. It also provides some simple code examples.You'll almost certainly need to refer to more substantial Xlib documentation (such as the O'Reilly volumes), as well as the *OpenGL Reference Manual*. The glXIntro reference page is a good starting point.

This section discusses the most important aspects of using Xlib and GLX and also provides several example programs in the following sections:

- "Getting Started With Xlib and GLX"
- "Using X Color Maps" on page 105
- "Using X Color Maps" on page 105
- "Using X Events" on page 106

Getting Started With Xlib and GLX

Note: Another example of using XLib is included in *OpenGL on Silicon Graphics Systems*.

To port your OpenGL code to use Xlib and GLX calls, follow these steps:

1. Add the necessary include files to your program. (See "Header Files" on page 18 for information on what files to include.)
2. Open a connection to a display: **XOpenDisplay()**.
3. Choose an X visual: **glXChooseVisual()**.
4. Create a GLX context: **glXCreateContext()**.
5. Create an X window or pixmap: **XCreateWindow()**.
6. Connect the GLX context to the X window: **glXMakeCurrent()**.

Opening a Window With GLX

Example 4-3 shows a simple way of following the steps given in the previous section.You can find a version of this code in the glXIntro reference page. This sample is more heavily commented than the one in the reference page and contains some additional examples.

Example 4-3 OpenGL and GLX Program

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

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

static Bool WaitForNotify(Display *d, XEvent *e, char *arg)
    { return(e->type == MapNotify) && (e->xmap.window == (Window)arg); }

int main(int argc, char**argv)
{
    Display *dpy;
    XVisualInfo *vi;
    Colormap cmap;
    XSetWindowAttributes swa;
    Window win;
    GLXContext cx;
    XEvent event;

/* get a connection   */
    dpy   = XOpenDisplay(0);
    if (!dpy) {
        fprintf(stderr, "Cannot open display.\n");
        exit(-1);
    }

/* get an appropriate visual */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy),
         attributeList);
    if (!vi) {
        fprintf(stderr, "Cannot find visual with desired attributes.\n");
        exit(-1);
    }

/* create a GLX context */
```

```
    cx = glXCreateContext(dpy, vi, 0, GL_FALSE);
    if (!cx) {
        fprintf(stderr, "Cannot create context.\n");
        exit(-1);
    }

/* create a colormap -- AllocAll for color index mode */
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
           vi->visual, AllocNone);
    if (!cmap) {
        fprintf(stderr, "Cannot allocate colormap.\n");
        exit(-1);
    }

    /* create a   window */
    swa.colormap = cmap;
    swa.border_pixel = 0;
    /* connect the context to the window */
    glXMakeCurrent(dpy, win, cx);

/* clear the buffer */
    glClearColor(1,1,0,1);
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();

/wait for a while */
    sleep(10);
/* exit cleanly */
    XCloseDisplay(dpy);
    exit(0);
}
```

Using X Color Maps

Here's a brief example of OpenGL GLX code that demonstrates the use of color maps:

```
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);
/* Note: if you don't say AllocAll, you can't load */
```

```
/* the color maps! */
    ...
if (index < visual->colormap_size) {
    xc.pixel = index;
    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);
}
```

Using X Events

Here's a simple example of a program that uses Xlib and OpenGL GLX calls for event handling:

```
swa.event_mask = ExposureMask | StructureNotifyMask
                | KeyPressMask | KeyReleaseMask;
do {
    XNextEvent(dpy, &event);
    switch (event.type) {
        case Expose:
            doRedraw = GL_TRUE;
            break;
        case ConfigureNotify:
            width = event.xconfigure.width;
            height = event.xconfigure.height;
            doRedraw = GL_TRUE;
            break;
        case KeyPress:
        {
            char buf[100];
            int rv;
            KeySym ks;

            rv = XLookupString(&event.xkey, buf, sizeof(buf), &ks, 0);
            switch (ks) {
                case XK_s:
                case XK_S:
                    doSave = GL_TRUE;
                    break;
                case XK_Escape:
                    return 0;
```

```
                    break;
                }
            }
        }
    } while (XPending(dpy));
```

Appendix A

OpenGL Commands and Their IRIS GL Equivalents

Table A-1 contains a list of equivalent calls that you might find useful while porting. The first column is an alphabetical list of IRIS GL calls, the second column contains the corresponding calls to use with OpenGL, and the third column contains pointers to any relevant discussion in the text.

Note: In many cases the OpenGL commands listed will function somewhat differently from the IRIS GL commands, and the parameters may be different as well.

Be sure to refer to the OpenGL reference pages in the *OpenGL Reference Manual* for detailed descriptions of the functions of the these commands and the parameters they take.

You might also need to refer to X or IRIS IM documentation; some appropriate X and IRIS IM manuals are listed in "Where to Get More Information" on page xiv.

Table A-1 IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
acbuf()	glAccum()	"Accumulation Buffer Calls" on page 61
acsize()	glXChooseVisual()	"Accumulation Buffer Calls" on page 61
addtopup()	glutCreateMenu, glutAddmenuEntry, glutAttachmenu, or use X or IRIS IM for menus	"GLX and GLUT Documentation" on page xvi Chapter 4, glXIntro reference page, X documentation, IRIS IM documentation
afunction()	glAlphaFunc()	"afunction() Test Functions" on page 60

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
arc(), arcf()	gluPartialDisk()[a]	"Editing toogl Output: An Example" on page 15 and "Porting Arcs and Circles" on page 42
backbuffer()	glDrawBuffer(GL_BACK)	glDrawBuffer reference page
backface()	glCullFace(GL_BACK)	glCullFace reference page
bbox2()	Not supported	"Porting bbox2() Calls" on page 65
bgnclosedline()	glBegin(GL_LINE_LOOP)	"Porting bgn/end Commands" on page 33 and "Porting Lines" on page 36
bgncurve()	gluBeginCurve()	"NURBS Curves" on page 53
bgnline()	glBegin(GL_LINE_STRIP)	"Porting bgn/end Commands" on page 33 and "Porting Lines" on page 36
bgnpoint()	glBegin(GL_POINTS)	"Porting bgn/end Commands" on page 33 and "Porting Points" on page 35
bgnpolygon()	glBegin(GL_POLYGON)	"Porting bgn/end Commands" on page 33, "Porting Polygons and Quadrilaterals" on page 37, and "Porting Tessellated Polygons" on page 41
bgnqstrip()	glBegin(GL_QUAD_STRIP)	"Porting bgn/end Commands" on page 33 and "Porting Polygons and Quadrilaterals" on page 37
bgnsurface()	gluBeginSurface()	"NURBS Surfaces" on page 54
bgntmesh()	glBegin(GL_TRIANGLE_STRIP)	"Porting bgn/end Commands" on page 33 and "Porting Triangles" on page 41
bgntrim()	gluBeginTrim()	"Trimming Curves" on page 54
blankscreen()	Use X for windowing.	Chapter 4 and glXIntro reference page

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
blanktime()	Use X for windowing.	Chapter 4 and glXIntro reference page
blendcolor()	glBlendColorEXT()	"Porting RealityEngine Graphics Features" on page 83
blendfunction()	glBlendFunc()	"Blending" on page 59
blink()	Use GLUT or X for color maps.	"GLX and GLUT Documentation" on page xvi, Chapter 4 and glXIntro reference page
blkqread()	Use GLUT or X for event handling.	"GLX and GLUT Documentation" on page xvi, Chapter 4 and glXIntro reference page
c3*(), c4*()	glColor*()	"Porting Color, Shading, and Writemask Commands" on page 44
callfunc()	Not supported.	"Porting Display Lists" on page 63
callobj()	glCallList()	"Porting bgn/end Commands" on page 33 and "Porting Display Lists" on page 63
charstr()	glCallLists()	"Fonts and Strings" on page 92
chunksize()	Not needed.	"Porting Display Lists" on page 63
circ(), circf()	gluDisk()	"Porting Arcs and Circles" on page 42
clear()	glClear(GL_COLOR_BUFFER_BIT)	"Windowing, Device, and Event Calls" on page 11 and "Porting Screen and Buffer Clearing Commands" on page 23
clearhitcode()	Not supported	"Porting Picking Calls" on page 79
clipplane()	glClipPlane()	"Porting Clipping Planes" on page 30
clkon()	XChangeKeyboardControl()	See X documentation.
clkoff()	XChangeKeyboardControl()	See X documentation.

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
closeobj()	glEndList()	"Porting Display Lists" on page 63
cmode()	glutInitDisplayMode, glXChooseVisual()	GLX and GLUT Documentation, Chapter 4 and glXIntro and glXChooseVisual() reference pages
cmov(), cmov2()	glRasterPos3()[a], glRasterPos2()[a]	"Porting Pixel Operations" on page 46
color(), colorf()	glIndex*()	"Porting bgn/end Commands" on page 33 and "Porting Color, Shading, and Writemask Commands" on page 44
compactify()	Not needed.	
concave()	gluBeginPolygon()[a]	
convolve()	glConvolutionFilter2DEXT(), glSeparableFilter2DEXT(), glConvolutionParameterEXT(), glPixelTransfer()	"Porting RealityEngine Graphics Features" on page 83
cpack()	glColor*()[a]	"Porting bgn/end Commands" on page 33 and "Porting Color, Shading, and Writemask Commands" on page 44
crv()	Not supported.	"Porting Curve and Surface Commands" on page 52
crvn()	Not supported.	"Porting Curve and Surface Commands" on page 52
curorigin()	Use X for cursors.	Chapter 4, glXIntro reference page, X documentation
cursoff()	Use X for cursors.	Chapter 4, glXIntro reference page, X documentation
curson()	Use X for cursors.	Chapter 4, glXIntro reference page, X documentation

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
curstype()	glutSetCursor, or use X for cursors	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
curvebasis()	glMap1()	"Porting Curve and Surface Commands" on page 52
curveit()	glEvalMesh1()	"Porting Curve and Surface Commands" on page 52
curveprecision()	Not supported.	"Porting Curve and Surface Commands" on page 52
cyclemap()	Use GLUT or use X for color maps.	"GLX and GLUT Documentation" on page xvi, Chapter 4, and glXIntro reference page
czclear()	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)	"Porting Screen and Buffer Clearing Commands" on page 23
dbtext()	Not supported	IRIS GL Dial and Button Box documentation
defbasis()	glMap1()	"Porting Curve and Surface Commands" on page 52
defcursor()	glutSetCursor, or use X for cursors	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
deflinestyle()	glLineStipple()	"Porting Lines" on page 36 and "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67
defpattern()	glPolygonStipple()	"Porting Polygons and Quadrilaterals" on page 37 and "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67
defpup()	Use GLUT or use X for menus.	"GLX and GLUT Documentation" on page xvi, Chapter 4, glXIntro reference page, X documentation

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
defrasterfont()	GLUT font rendering functions, glXUseXFont()[a]	"GLX and GLUT Documentation" on page xvi, "Fonts and Strings" on page 92
delobj()	glDeleteLists()	"Porting Display Lists" on page 63
deltag()	Not supported.	"Porting Display Lists" on page 63
depthcue()	glFog()[a]	"Porting Depth Cueing and Fog Commands" on page 48
dglclose()	Not needed—OpenGL is network transparent.	
dglopen()	Not needed—OpenGL is network transparent.	
displacepolygon()	glPolygonOffsetEXT() glPolygonOffset()—OpenGL 1.1	"Porting RealityEngine Graphics Features" on page 83
dither()	glEnable(GL_DITHER)	"Porting Color, Shading, and Writemask Commands" on page 44
dopup()	Use GLUT or use X for menus.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
doublebuffer()	glXChooseVisual()	Chapter 4 and glXIntro reference page
draw()	glBegin(GL_LINES)[a]	"Porting Commands That Required Current Graphics Positions" on page 22 and "Porting Lines" on page 36
drawmode()	glXMakeCurrent()[a]	
editobj()	Not supported.	"Porting Display Lists" on page 63
endclosedline()	glEnd()	"Porting bgn/end Commands" on page 33 and "Porting Lines" on page 36

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
endcurve()	gluEndCurve()	"Porting Curve and Surface Commands" on page 52
endfeedback()	glRenderMode(GL_RENDER)	"Porting Feedback Calls" on page 80
endfullscreen()	Not supported.	
endline()	glEnd()	"Porting bgn/end Commands" on page 33
endpick()	glRenderMode(GL_RENDER)	"Porting Picking Calls" on page 79
endpoint()	glEnd()	"Porting bgn/end Commands" on page 33 and "Porting Points" on page 35
endpolygon()	glEnd()	"Porting bgn/end Commands" on page 33 and "Porting Polygons and Quadrilaterals" on page 37
endpupmode()	Use GLUT or use X for menus.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
endqstrip()	glEnd()	"Porting bgn/end Commands" on page 33 and "Porting Polygons and Quadrilaterals" on page 37
endselect()	glRenderMode(GL_RENDER)	"Porting Picking Calls" on page 79
endsurface()	gluEndSurface()	"NURBS Surfaces" on page 54
endtmesh()	glEnd()	"Porting bgn/end Commands" on page 33 and "Porting Triangles" on page 41
endtrim()	gluEndTrim()	"Trimming Curves" on page 54
fbsubtexload()	Not supported in OpenGL 1.0. Use glSubTexture*() in OpenGL 1.1.	"Porting RealityEngine Graphics Features" on page 83
feedback()	glFeedbackBuffer()	"Porting Feedback Calls" on page 80
finish()	glFinish()	

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
fogvertex()	glFog()	"Porting Depth Cueing and Fog Commands" on page 48
font()	See glListBase().	
foreground()	glutSwapBuffers, glutPushWindow, glutPopWindow, or use X for windowing	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
freepup()	Use GLUT or X for menus.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
frontbuffer()	glDrawBuffer(GL_FRONT)	
frontface()	See glCullFace().	
fsubtexload()	glCopyTexSubImage2D()—OpenGL 1.1	
fudge()	Use X for windowing.	
fullscrn()	glutFullScreen	See GLX and GLUT Documentation
gammaramp()	Use GLUT or X for color maps.	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
gbegin()	Use X for windowing.	Chapter 4 and glXIntro reference page
gconfig()	No equivalent (not needed).	Chapter 4 and glXIntro reference page
genobj()	glGenLists()	"Porting Display Lists" on page 63
gentag()	Not supported.	
getbackface()	glGet*()	"Porting IRIS GL get* Commands" on page 20
getbuffer()	glGet*()	"Porting IRIS GL get* Commands" on page 20

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
getbutton()	Use X for windowing.	"Porting IRIS GL get* Commands" on page 20, Chapter 4, and glXIntro reference page
getcmmode()	glXGetCurrentContext()	"Porting IRIS GL get* Commands" on page 20, Chapter 4 and "Porting Color, Shading, and Writemask Commands" on page 44
getcolor()	glGet*()	
getcpos()	glGet*()	"Porting IRIS GL get* Commands" on page 20
getcursor()	Not supported.	"Porting IRIS GL get* Commands" on page 20
getdcm()	glIsEnabled()	"Porting IRIS GL get* Commands" on page 20 and "Porting Depth Cueing and Fog Commands" on page 48
getdepth()	glGet*()	"Porting IRIS GL get* Commands" on page 20
getdescender()	Use X for fonts.	"Fonts and Strings" on page 92 and "Porting IRIS GL get* Commands" on page 20
getdev()	Not supported.	"Porting IRIS GL get* Commands" on page 20
getdisplaymode()	glGet*()	"Porting IRIS GL get* Commands" on page 20
getdrawmode()	glXGetCurrentContext()	"Porting IRIS GL get* Commands" on page 20
getfont()	Use GLUT or X for fonts.	GLX and GLUT Documentation, "Porting IRIS GL get* Commands" on page 20 and "Fonts and Strings" on page 92

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
getgdesc()	glGet*(), glXGetConfig(), glXGetCurrentContext(), glXGetCurrentDrawable()	"Porting IRIS GL get* Commands" on page 20
getgpos()	Not supported.	"Porting Commands That Required Current Graphics Positions" on page 22 and "Porting IRIS GL get* Commands" on page 20
getheight()	Use X for fonts.	GLX and GLUT Documentation, "Fonts and Strings" on page 92 and "Porting IRIS GL get* Commands" on page 20
gethgram()	glGetHistogramEXT()	"Porting RealityEngine Graphics Features" on page 83
gethitcode()	Not supported.	"Porting Picking Calls" on page 79 and "Porting IRIS GL get* Commands" on page 20
getlsbackup()	Not supported.	"Porting Lines" on page 36 and "Porting IRIS GL get* Commands" on page 20
getlsrepeat()	glGet*()	"Porting IRIS GL get* Commands" on page 20 and "Porting Lines" on page 36
getlstyle()	glGet*()	"Porting IRIS GL get* Commands" on page 20 and "Porting Lines" on page 36
getlwidth()	glGet*()	"Porting IRIS GL get* Commands" on page 20 and "Porting Lines" on page 36
getmap(void)	Not supported.	"Porting IRIS GL get* Commands" on page 20, Chapter 4 and "Porting Color, Shading, and Writemask Commands" on page 44

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
getmatrix()	glGet*(GL_MODELVIEW_MATRIX), glGet*(GL_PROJECTION_MATRIX)	"Porting IRIS GL get* Commands" on page 20 and "Porting Matrix and Transformation Calls" on page 24
getmcolor()	Not supported.	"Porting IRIS GL get* Commands" on page 20, "Porting Color, Shading, and Writemask Commands" on page 44, Chapter 4 and glXIntro reference page
getminmax()	glGetMinmaxEXT()	"Porting RealityEngine Graphics Features" on page 83
getmmode()	glGet*(GL_MATRIX_MODE)	"Porting get* Calls for Matrices and Transformations" on page 29.
getmonitor()	Not supported.	"Porting IRIS GL get* Commands" on page 20
getnurbsproperty()	gluGetNurbsProperty()	"Porting IRIS GL get* Commands" on page 20
getopenobj()	Not supported.	"Porting Display Lists" on page 63 and "Porting IRIS GL get* Commands" on page 20
getorigin()	Use X for windowing.	"Porting IRIS GL get* Commands" on page 20 and Chapter 4 and glXIntro reference page
getpattern()	glGetPolygonStipple()	"Porting IRIS GL get* Commands" on page 20 and "Porting Polygons and Quadrilaterals" on page 37
getplanes()	glGet*(GL_RED_BITS), glGet*(GL_GREEN_BITS), glGet*(GL_BLUE_BITS)	"Porting IRIS GL get* Commands" on page 20
getport()	Use X for windowing.	"Porting IRIS GL get* Commands" on page 20, Chapter 4, and glXIntro reference page

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
getresetls()	Not supported.	"Porting Lines" on page 36 and "Porting IRIS GL get* Commands" on page 20
getscrbox()	Not supported.	"Porting IRIS GL get* Commands" on page 20 and "Porting Viewports, Screenmasks, and Scrboxes" on page 30
getscrmask()	glGet*(GL_SCISSOR_BOX)	"Porting IRIS GL get* Commands" on page 20 and "Porting Viewports, Screenmasks, and Scrboxes" on page 30
getshade()	glGet*(GL_CURRENT_INDEX)	"Porting IRIS GL get* Commands" on page 20
getsize()	Use X for windowing.	"Porting IRIS GL get* Commands" on page 20, Chapter 4, and glXIntro reference page
getsm()	glGet*(GL_SHADE_MODEL)	"Porting IRIS GL get* Commands" on page 20 and "Porting Color, Shading, and Writemask Commands" on page 44
getvaluator()	Use glutMainLoop() or use X for event handling.	GLX and GLUT Documentation, "Porting IRIS GL get* Commands" on page 20, Chapter 4, and glXIntro reference page
getvideo()	Not supported.	"Porting IRIS GL get* Commands" on page 20
getviewport()	glGet*(GL_VIEWPORT)	"Porting IRIS GL get* Commands" on page 20 and "Porting Viewports, Screenmasks, and Scrboxes" on page 30
getwritemask()	glGet*(GL_INDEX_WRITEMASK)	"Porting IRIS GL get* Commands" on page 20 and "Porting Color, Shading, and Writemask Commands" on page 44

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
getwscrn()	Use X for windowing.	"Porting IRIS GL get* Commands" on page 20, Chapter 4 and glXIntro reference page
getzbuffer()	glIsEnabled(GL_DEPTH_TEST)	"Porting IRIS GL get* Commands" on page 20
gexit()	Use X for windowing.	
gflush()	glFlush()	
ginit()	Use GLUT or use X for windowing.	GLX and GLUT Documentation, Chapter 4, and glXIntro reference page
glcompat()	Not supported.	
GLXgetconfig()	glXChooseVisual(), glXGetConfig()	Chapter 4 and glXIntro reference page
GLXlink()	Combination of glXCreateContext() and glXMakeCurrent()	Chapter 4 and glXIntro reference page
GLXunlink()	glXMakeCurrent(display_name, None, NULL)	Chapter 4 and glXIntro reference page
GLXwinset()	glXMakeCurrent() has some of the functionality.	Chapter 4 and glXIntro reference page
greset()	Not supported.	"Porting greset()" on page 19
gRGBcolor()	glGet*(GL_CURRENT_ RASTER_COLOR)	"Porting Color, Shading, and Writemask Commands" on page 44
gRGBcursor()	Use GLUT or use X for cursors.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
gRGBmask()	glGet*(GL_COLOR_WRITEMASK)	"Porting Color, Shading, and Writemask Commands" on page 44
gselect()	glSelectBuffer()	

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
gsync()	Use GLUt or use X for windowing.	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
gversion()	glGetString(GL_RENDERER)[a]	Chapter 4 and glXIntro reference page
hgram()	glHistogramEXT(), glResetHistogramEXT()	"Porting RealityEngine Graphics Features" on page 83
iconsize()	glutIconifyWindow, or use X	GLX and GLUT Documentation or X documentation for XIconSize()
icontitle()	glutSetIconTitle, or use X	GLX and GLUT Documentation or X documentation for XSetIconName()
ilbuffer()	Not supported.	"Porting RealityEngine Graphics Features" on page 83
ildraw()	Not supported.	"Porting RealityEngine Graphics Features" on page 83
imakebackground()	Use X for event handling.	Chapter 4 and glXIntro reference page
initnames()	glInitNames()	
ismex()	Not supported.	
isobj()	glIsList()	"Porting Display Lists" on page 63
isqueued()	Use X for event handling.	Chapter 4 and glXIntro reference page
istag()	Not supported.	"Stencil Plane Calls" on page 63
istexloaded()	glAreTexturesResident() — OpenGL 1.1	"Porting RealityEngine Graphics Features" on page 83
keepaspect()	Use X for windowing.	Chapter 4 and glXIntro reference page
lampoff()	Not supported.	See X documentation for XChangeKeyboardControl().

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
lampon()	Not supported.	See X documentation for XChangeKeyboardControl().
leftbuffer()	glDrawBuffer(GL_LEFT)	"Porting RealityEngine Graphics Features" on page 83
linesmooth()	glEnable(GL_LINE_SMOOTH)	"Porting Lines" on page 36 and "Antialiasing Calls" on page 60
linewidth() linewidthf*()	glLineWidth()	"Porting Lines" on page 36
lmbind()	glEnable(GL_LIGHTING) glEnable(GL_LIGHTi)	"Porting bgn/end Commands" on page 33, "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67, and "Porting Lighting and Materials Calls" on page 68
lmcolor()	glColorMaterial()	"Porting Lighting and Materials Calls" on page 68
lmdef()	glMaterial() glLight() glLightModel()	"Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67 and "Porting Lighting and Materials Calls" on page 68
loadmatrix()	glLoadMatrix()	"Porting Matrix and Transformation Calls" on page 24
loadname()	glLoadName()	"Porting Picking Calls" on page 79
logicop()	glLogicOp()	"Porting Pixel Operations" on page 46
lookat()	gluLookAt()[a]	"Porting Matrix and Transformation Calls" on page 24
lrectread()	glReadPixels()	"Porting Pixel Operations" on page 46
lrectwrite()	glDrawPixels()	"Porting Pixel Operations" on page 46

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
lRGBrange()	Not supported; see glFog().	"Porting Depth Cueing and Fog Commands" on page 48
lsbackup()	Not supported.	"Porting Lines" on page 36
lsetdepth()	glDepthRange()	"Porting Depth Cueing and Fog Commands" on page 48
lshaderange()	Not supported; see glFog().	"Porting Depth Cueing and Fog Commands" on page 48
lsrepeat()	glLineStipple()	"Porting Lines" on page 36
makeobj()	glNewList()	"Porting Display Lists" on page 63
maketag()	Not supported.	"Stencil Plane Calls" on page 63
mapcolor()	XStoreColor()	Chapter 4
mapw()	gluProject()	"Porting Matrix and Transformation Calls" on page 24
maxsize()	Use GLUT or use X for windowing.	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
minmax()	glMinmaxEXT()	"Porting RealityEngine Graphics Features" on page 83
minsize()	Use GLUT or use X for windowing.	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
mmode()	glMatrixMode()	"Porting Matrix and Transformation Calls" on page 24
monobuffer()	Superseded by selection of an appropriate GLX visual.	glXChooseVisual() reference page and "Porting RealityEngine Graphics Features" on page 83
move()	Not supported.	"Porting Commands That Required Current Graphics Positions" on page 22

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
msalpha()	glEnable(GL_SAMPLE_ALPHA_TO_MASK_SGIS), glEnable(GL_SAMPLE_ALPHA_TO_ONE_SGIS)	"Porting RealityEngine Graphics Features" on page 83
msmask()	glSampleMaskSGIS()	"Porting RealityEngine Graphics Features" on page 83
mspattern()	glSamplePatternSGIS()	"Porting RealityEngine Graphics Features" on page 83
mssize()	glXChooseVisual with attribute GLX_SAMPLE_BUFFERS_SGIS	"Porting RealityEngine Graphics Features" on page 83
mswapbuffers()	glutSwapBuffers, glxSwapbuffers	GLX and GLUT Documentation or X documentation
multimap()	Use X for color maps.	"Porting Color, Shading, and Writemask Commands" on page 44, also Chapter 4, glXIntro reference page
multisample()	glEnable(GL_MULTISAMPLE_SGIS)	"Porting RealityEngine Graphics Features" on page 83
multmatrix()	glMultMatrix()	
n3f()	glNormal3fv()	"Porting bgn/end Commands" on page 33
newpup()	Use GLUT or X for menus.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
newtag()	Not supported.	"Porting Display Lists" on page 63
nmode()	glEnable(GL_NORMALIZE)	
noborder()	Use X for windowing.	Chapter 4 and glXIntro reference page
noise()	Use X for event handling.	Chapter 4 and glXIntro reference page

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
noport()	Use X for windowing.	Chapter 4 and glXIntro reference page
normal()	glNormal3fv()	
nurbscurve()	gluNurbsCurve()[a]	"NURBS Curves" on page 53 and "Trimming Curves" on page 54
nurbssurface()	gluNurbsSurface()[a]	"NURBS Surfaces" on page 54
objdelete()	Not supported.	"Stencil Plane Calls" on page 63
objinsert()	Not supported.	"Stencil Plane Calls" on page 63
objreplace()	Not supported.	"Porting Display Lists" on page 63
onemap()	Use GLUT or X for color maps.	"GLX and GLUT Documentation" on page xvi, "Porting Color, Shading, and Writemask Commands" on page 44, and Chapter 4 and glXIntro reference page
ortho()	glOrtho()	"Porting Matrix and Transformation Calls" on page 24
ortho2()	gluOrtho2D()	"Porting Matrix and Transformation Calls" on page 24
overlay()	Use GLUT overlay functions or use X.	"GLX and GLUT Documentation" on page xvi, Chapter 4, glXIntro reference pages, and glXChooseVisual()
pagecolor()	Not supported.	
passthrough()	glPassThrough()	"Porting Feedback Calls" on page 80
patch()	glEvalMesh2()[a]	"Porting Curve and Surface Commands" on page 52
patchbasis()	glMap2()[a]	"Porting Curve and Surface Commands" on page 52

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
patchcurves()	glMap2()[a]	"Porting Curve and Surface Commands" on page 52
patchprecision()	Not supported.	"Porting Curve and Surface Commands" on page 52
pclos()	Not supported; see glEnd().	"Porting Commands That Required Current Graphics Positions" on page 22 and "Porting Polygons and Quadrilaterals" on page 37
pdr()	Not supported; see glVertex().	"Porting Commands That Required Current Graphics Positions" on page 22 and "Porting Polygons and Quadrilaterals" on page 37
perspective()	gluPerspective()	"Porting Matrix and Transformation Calls" on page 24
pick()	gluPickMatrix()[a] and glRenderMode(GL_SELECT)	"Porting Picking Calls" on page 79
picksize()	gluPickMatrix()	"Porting Matrix and Transformation Calls" on page 24 and "Porting Picking Calls" on page 79
pixelmap()	glPixelMap()	"Porting RealityEngine Graphics Features" on page 83
pixeltransfer()	glPixelTransfer()	"Porting RealityEngine Graphics Features" on page 83
pixmode()	glPixelTransfer() and glPixelStore()	"Porting Pixel Operations" on page 46
pmv()	Not supported; see glBegin() and glVertex().	"Porting Commands That Required Current Graphics Positions" on page 22 and "Porting Polygons and Quadrilaterals" on page 37
pnt*()	glBegin(GL_POINTS)[a]	"Porting Points" on page 35
pntsize(), pntsizef()	glPointSize()	"Porting Points" on page 35

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
pntsmooth()	glEnable(GL_POINT_SMOOTH)	"Porting Points" on page 35 and "Antialiasing Calls" on page 60
polarview()	Not supported; see glRotate() and glTranslate().	"Porting Matrix and Transformation Calls" on page 24
polf()	Not supported.	"Porting Polygons and Quadrilaterals" on page 37
poly()	Not supported.	"Porting Polygons and Quadrilaterals" on page 37
polymode()	glPolygonMode()	"Porting Polygons and Quadrilaterals" on page 37
polysmooth()	glEnable(GL_POLYGON_SMOOTH)	"Antialiasing Calls" on page 60
popattributes()	glPopAttrib(), glPopClientAttrib()	"Porting greset()" on page 19 explains how to use glPopAttrib()
popmatrix()	glPopMatrix()	"Porting Matrix and Transformation Calls" on page 24
popname()	glPopName()	"Porting Picking Calls" on page 79
popviewport()	glPopAttrib()	"Porting Viewports, Screenmasks, and Scrboxes" on page 30
prefposition()	Use X for windowing.	Chapter 4 and glXIntro reference page
prefsize()	Use X for windowing.	Chapter 4 and glXIntro reference page
pupmode()	Use X for windowing.	Chapter 4 and glXIntro reference page
pushattributes()	glPushAttrib(), glPushClientAttrib()	
pushmatrix()	glPushMatrix()	"Porting Matrix and Transformation Calls" on page 24
pushname()	glPushName()	"Porting Picking Calls" on page 79

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
pushviewport()	glPushAttrib(GL_VIEWPORT)	"Porting Viewports, Screenmasks, and Scrboxes" on page 30
pwlcurve()	gluPWLCurve()	"Trimming Curves" on page 54
qcontrol()	Use X for event handling.	Chapter 4 and glXIntro reference page
qdevice()	Use X for event handling.	Chapter 4 and glXIntro reference page
qenter()	Use X for event handling.	Chapter 4 and glXIntro reference page
qgetfd()	Use X for event handling.	Chapter 4 and glXIntro reference page
qread()	Use X for event handling.	Chapter 4 and glXIntro reference page
qreset()	Use X for event handling.	Chapter 4 and glXIntro reference page
qtest()	Use X for event handling.	Chapter 4 and glXIntro reference page
rcrv(), rcrvn()	Not supported.	"Porting Curve and Surface Commands" on page 52
rdr()	Not supported.	"Porting Commands That Required Current Graphics Positions" on page 22
readcomponent()	glReadPixels() gives partial support; some readcomponent() features aren't supported.	"Porting RealityEngine Graphics Features" on page 83
readdisplay()	Not supported.	
readpixels()	glReadPixels()	"Porting Pixel Operations" on page 46
readRGB()	Not supported.	"Porting Pixel Operations" on page 46

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
readsource()	glReadBuffer()	"Porting Pixel Operations" on page 46
rect(), rectf()	See glRect() and glPolygonMode().	"Porting Polygons and Quadrilaterals" on page 37
rectcopy()	glCopyPixels()	"Porting Pixel Operations" on page 46
rectread()	glReadPixels()	"Porting Pixel Operations" on page 46
rectwrite()	glDrawPixels()	"Porting Pixel Operations" on page 46
rectzoom()	glPixelZoom()	"Porting Pixel Operations" on page 46
resetls()	Not supported.	"Porting Lines" on page 36
reshapeviewport()	Not supported.	Chapter 4 and glXIntro reference page
RGBcolor()	glColor()	"Porting bgn/end Commands" on page 33 and "Porting Color, Shading, and Writemask Commands" on page 44
RGBcursor()	Use glutSetCursor or use X for cursors.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
RGBmode()	Use X for windowing.	Chapter 4 and glXIntro reference page
RGBrange()	Not supported.	
RGBsize()	Not supported, but the glXChooseVisual() function does some similar things.	glXChooseVisual() reference page
RGBwritemask()	glColorMask()	"Porting Color, Shading, and Writemask Commands" on page 44

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
rightbuffer()	glDrawBuffer(GL_RIGHT)	"Porting RealityEngine Graphics Features" on page 83
ringbell()	Not supported.	X documentation for XBell()
rmv()	Not supported.	"Porting Commands That Required Current Graphics Positions" on page 22
rot()	glRotate()	"Porting Matrix and Transformation Calls" on page 24
rotate()	glRotate()	"Porting Matrix and Transformation Calls" on page 24
rpatch()	Not supported.	"Porting Curve and Surface Commands" on page 52
rpdr()	Not supported.	"Porting Commands That Required Current Graphics Positions" on page 22 and "Porting Polygons and Quadrilaterals" on page 37
rpmv()	Not supported.	"Porting Commands That Required Current Graphics Positions" on page 22 and "Porting Polygons and Quadrilaterals" on page 37
sbox(), sboxf()	glRect()[a]	"Porting Polygons and Quadrilaterals" on page 37
scale()	glScale()	"Porting Matrix and Transformation Calls" on page 24
sclear()	glClear(GL_STENCIL_BUFFER_BIT)	"Porting Screen and Buffer Clearing Commands" on page 23 and "Stencil Plane Calls" on page 63
scrbox()	Not supported.	"Porting Viewports, Screenmasks, and Scrboxes" on page 30
screenspace()	Not supported.	"Porting Matrix and Transformation Calls" on page 24

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
scrmask()	glScissor()	"Porting Viewports, Screenmasks, and Scrboxes" on page 30
scrnattach()	Use X for windowing.	Chapter 4 and glXIntro reference page
scrnselect()	Use X for windowing.	Chapter 4 and glXIntro reference page
scrsubdivide()	Not supported.	
select()	glRenderMode()	"Porting Picking Calls" on page 79
setbell()	Not supported.	X documentation for XChangeKeyboardControl()
setcursor()	glutSetCursor, or use X for cursors	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, and X documentation
setdblights()	Not supported.	dial and button box documentation
setdepth()	glDepthRange()[a]	
setlinestyle()	glLineStipple()	"Porting Lines" on page 36 and "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67
setmap()	Use GLUT or X for color maps.	GLX and GLUT Documentation, "Porting Color, Shading, and Writemask Commands" on page 44, Chapter 4, and glXIntro reference page
setmonitor()	Not supported.	
setnurbsproperty()	gluNurbsProperty()	
setpattern()	glPolygonStipple()	"Porting Polygons and Quadrilaterals" on page 37 and "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
setpup()	Use GLUT or X for menus.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
setvaluator()	Use GLUT or X for devices.	GLX and GLUT Documentation, Chapter 4, glXIntro reference page, X documentation
setvideo()	Not supported.	
shademodel()	glShadeModel()	"Porting Color, Shading, and Writemask Commands" on page 44
shaderange()	glFog()	
singlebuffer()	Use X for windowing.	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
smoothline()	glEnable(GL_LINE_SMOOTH)	"Porting Lines" on page 36
spclos()	Not supported.	"Porting Polygons and Quadrilaterals" on page 37
splf()	Not supported see glBegin().	"Porting Polygons and Quadrilaterals" on page 37
stencil()	glStencilFunc(), glStencilOp()	"Stencil Plane Calls" on page 63
stensize()	glStencilMask()	"Stencil Plane Calls" on page 63
stepunit()	Use X for windowing.	Chapter 4 and glXIntro reference page
stereobuffer()	Superseded by selection of an appropriate GLX visual.	glXChooseVisual() reference page and "Porting RealityEngine Graphics Features" on page 83
strwidth()	Use X for fonts and strings.	"Fonts and Strings" on page 92
subpixel()	Not needed.	"Porting Antialiasing Calls" on page 58

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
subtexload()	glTexSubImage2DEXT()—OpenGL 1.0 glTexSubImage2D()—OpenGL 1.1	"Porting RealityEngine Graphics Features" on page 83
swapbuffers()	glXSwapBuffers()	glXIntro and glXSwapBuffers() reference pages
swapinterval()	Not supported.	
swaptmesh()	Not supported; see glBegin(GL_TRIANGLE_FAN)	"Porting Triangles" on page 41
swinopen()	Use X for windowing	Chapter 4 and glXIntro reference page
swritemask()	glStencilMask()	"Stencil Plane Calls" on page 63
t2*(), t3*(), t4*()	glTexCoord*()	"Porting Texture Calls" on page 73
tevbind()	glTexEnv()	"Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67 and "Porting Lighting and Materials Calls" on page 68
tevdef()	glTexEnv()	"Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67, "Porting Lighting and Materials Calls" on page 68, and "Translating tevdef()" on page 75
texbind()	glTexImage2D(), glTexParameter(), gluBuild2DMipmaps(),	"Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67 and "Porting Texture Calls" on page 73
texdef2d()	glTexImage2D(), glTexParameter(), gluBuild2DMipmaps()	"Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67, "Porting Lighting and Materials Calls" on page 68, and "Translating texdef()" on page 76

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
texdef3d()	glTexImage3DEXT()	"Porting RealityEngine Graphics Features" on page 83
texgen()	glTexGen()	"Porting Lighting and Materials Calls" on page 68 and "Translating texgen()" on page 78
textcolor()	Not supported.	
textinit()	Not supported.	
textport()	Not supported.	
tie()	Use X for event handling.	Chapter 4 and glXIntro reference page
tlutbind()	Not supported.	"Porting RealityEngine Graphics Features" on page 83
tlutdef()	Not supported.	"Porting RealityEngine Graphics Features" on page 83
tpoff()	Not supported.	
tpon()	Not supported.	
translate()	glTranslate()	"Porting Matrix and Transformation Calls" on page 24
underlay()	glXChooseVisual()	
unqdevice()	Use X for event handling.	Chapter 4 and glXIntro reference page
v2*(), v3*(), v4*()	glVertex*()	"Porting v() Commands" on page 33
videocmd()	Not supported.	
viewport()	glViewport()	"Porting Viewports, Screenmasks, and Scrboxes" on page 30
winattach()	Use GLUT or X for windowing.	GLX and GLUT Documentation, Chapter 4, and glXIntro reference page

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
winclose()	glXDestroyContext(), XCloseDisplay()	
winconstraints()	Use X for windowing.	Chapter 4 and glXIntro reference page
windepth()	Use X for windowing.	Chapter 4 and glXIntro reference page
window()	glFrustum()	"Porting Matrix and Transformation Calls" on page 24
winget()	glXGetCurrentContext()	
winmove()	glutPositionWindow(), or use X for windowing	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
winopen()	glutShowWindow(), or use X for windowing	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
winpop()	glutPopWindow, or use X for windowing	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
winposition()	glutPositionWindow, or use X for windowing	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
winpush()	glutPushWindow, or use X for windowing	GLX and GLUT Documentation, Chapter 4 and glXIntro reference page
winset()	Use GLUT or use X for windowing.	GLX and GLUT Documentation, Chapter 4 and glXIntro and glXMakeCurrent() reference pages
wintitle()	glutSetWindowTitle, or use X for windowing	GLX and GLUT Documentation Chapter 4 and glXIntro reference page
wmpack()	glColorMask()	Chapter 4

Table A-1 (continued) IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX Equivalent	Where Discussed
writemask()	glIndexMask()	
writepixels()	glDrawPixels()	
writeRGB()	glDrawPixels()	
xfpt*()	Not supported.	"Porting Picking Calls" on page 79; "Porting Feedback Calls" on page 80
zbsize()	Superseded by selection of an appropriate GLX visual.	glXChooseVisual() reference page and "Porting RealityEngine Graphics Features" on page 83
zbuffer()	glEnable(GL_DEPTH_TEST)	
zclear()	glClear(GL_DEPTH_BUFFER_BIT)	"Porting Screen and Buffer Clearing Commands" on page 23
zdraw()	Not supported.	
zfunction()	glDepthFunc()	
zsource()	Not supported.	
zwritemask()	glDepthMask()	"Porting Color, Shading, and Writemask Commands" on page 44

a. Note that this is not a direct equivalent of IRIS GL functionality—be careful when porting.

Appendix B

Differences Between OpenGL and IRIS GL

This appendix contains a list of differences between OpenGL and IRIS GL in alphabetical order. The list is based, in part, on a document by Kurt Akeley (May 1992) that was updated by Mark Kilgard in May 1997. Each difference is given a simple concept name, based on an IRIS GL concept, followed by a description. To look up a difference, find the concept name in the alphabetical list and read the description.

accumulation from the z-buffer	Reading from and writing to the z-buffer from the accumulation buffer isn't supported in OpenGL
accumulation wrapping	OpenGL accumulation buffer operation is not defined when component values exceed 1.0 or go below -1.0.
antialiased lines	OpenGL stipples antialiased lines. IRIS GL does not.
arc	OpenGL supports arcs in its utility library (GLU).
attribute lists	The attributes pushed by IRIS GL **pushattributes()** differ from any of the attribute sets that are pushed by OpenGL **glPushAttrib()** and **glPushClientAttrib()**. Because all OpenGL states can be read back, however, it is possible to implement any desired push/pop semantics using the OpenGL API.
automatic mipmap generation	The OpenGL texture interface does not support automatic mipmap generation. However, the utility library supports automatic generation of mipmap images for both 1D and 2D textures. See the gluBuild1DMipmaps and gluBuild2DMipmaps reference pages.
automatic texture scaling	The OpenGL texture interface does not support automatic scaling of images to power-of-two dimensions. However, scaling is supported by the OpenGL utility library. See the gluScaleImage reference page.
bbox	OpenGL does not support conditional execution of display lists.

buffers, multiple	Anything involving multiple buffers or windowing must be done in X or with GLUT.
callfunc	OpenGL does not support callback from display lists. Note that IRIS GL also did not support this functionality when client and server were on different platforms.
circle	OpenGL supports circles in the GLU. In OpenGL both circles and arcs (disks and partial disks) can have holes. Subdivision of the primitives can be changed in OpenGL and the primitives' surface normals are available for lighting.
clear options	OpenGL really clears buffers. It does not apply most currently specified pixel operations, such as blending and logicop, regardless of their modes, though there are some options that are applied. To clear using such features, you must render a window-size polygon.
closed lines	OpenGL renders all single-width aliased lines such that abutting lines share no pixels. This means that the "last" pixel of an independent line is not drawn.
color maps	Changing the color map under OpenGL must be done using the X color map.
color/normal flag	OpenGL lighting is explicitly enabled or disabled. When enabled, it is effective regardless of the order in which colors and normals are specified. Lighting cannot be enabled or disabled between OpenGL **glBegin()** and **glEnd()** commands. If you need to disable lighting between **glBegin()** and **glEnd()**, you must do it by specifying zero ambient, diffuse, and specular material reflectance. Then set the material emission to the desired color.
color packing	OpenGL has no need for **cpack()**; you can use four-item vectors to specify colors instead.

color range with c3i()	The OpenGL **glColor*()** routines that appear to correspond directly to IRIS GL **c3*()** and **c4*()** routines are not actually equivalent. For instance, the IRIS GL **c3i()** function took arguments in the range [0, 255] for each color; but in OpenGL, **glColor3i()** allows signed arguments with values up to over two billion. Check the *OpenGL Programming Guide* for details on argument value ranges, and use **glColor3ub()** as a replacement for **c3i()**.
concave polygons	The OpenGL API does not handle concave polygons, but the GLU library does provide support for decomposing concave, non-self-intersecting contours into triangles. These triangles can either be drawn immediately or returned. See the gluNewTess reference page.
current computed color	OpenGL has no notion of a current computed color. If you're using OpenGL as a lighting engine, you can use feedback to obtain colors generated by lighting calculations.
current graphics position	OpenGL does not maintain a current graphics position. IRIS GL commands that depended on current graphics position, such as relative lines and polygons, are not available in OpenGL.
curves	OpenGL does not support IRIS GL curves. Use of NURBS curves is recommended.
defs/binds	OpenGL 1.0 does not have the concept of a material, light, or texture objects, only of material, light, and texture properties. OpenGL programmers can use display lists to create their own objects. In OpenGL 1.1, you can use texture objects, as discussed in the *OpenGL Programming Guide*.
depthcue	OpenGL provides no direct support for depth cueing. However, its fog support is a more general capability that you can easily use to emulate IRIS GL **depthcue()**.

display list editing	OpenGL display lists cannot be edited, only created and destroyed. Because display list names are specified by the programmer, however, it is possible to redefine individual display lists in a hierarchy. OpenGL display lists are designed for data caching, not for database management. They are guaranteed to be stored on the server in client/server environments, so they are not limited by network bandwidth during execution. OpenGL display lists can be called between **glBegin()** and **glEnd()** commands, so the display list hierarchy can be made fine enough that it can, in effect, be edited.
error checking	OpenGL checks for errors more carefully than IRIS GL. For example, all OpenGL commands that are not accepted between **glBegin()** and **glEnd()** are detected as errors, and have no other effect.
error return values	When an OpenGL command that returns a value detects an error, it always returns zero. OpenGL commands that return data through passed pointers make no change to the array contents if an error is detected.
error side effects	When an OpenGL command results in an error, its only side effect is to update the error flag to the appropriate value. No other state changes are made. (An exception is the OUT_OF_MEMORY error, which is fatal.)
evaluators	Input and output that was done with such functions as **getbutton()**, **qread()**, and **qdevice()** in IRIS GL must be done using X calls with OpenGL, as must cursor-manipulation functions.
feedback	In OpenGL, feedback is standardized so that it doesn't change from machine to machine. "Porting Feedback Calls" on page 80 explains how to port your IRIS GL feedback calls.
fog	In OpenGL, you can't use depth-cueing and fog at the same time, because fog is used to emulate depth-cueing. IRIS GL allows more options to fog; some OpenGL implementations may compute fog per-vertex instead of per-fragment. Some new extension for fog functionality will be released as extensions to OpenGL 1.1.

fonts and strings	OpenGL expects character glyphs to be manipulated as individual display lists. It provides a display list calling function that accepts a list of display list names, each name represented as 1, 2, or 4 bytes. **glCallLists()** adds a separately specified offset to each display list name before the call, allowing lists of display list names to be treated as strings. This mechanism provides all the functionality of IRIS GL fonts, and considerably more. For example, characters consisting of triangles can be easily manipulated. OpenGL programs can use the OpenGL Character Renderer (GLC) library for accessing particular fonts. See the glcintro reference page.
frontbuffer	IRIS GL had complex rules for defeating rendering to the front buffer in singlebuffer mode. OpenGL does as it is asked in this regard.
hollow polygons	OpenGL does not support hollow polygons. However, you can use the OpenGL stencil capability to render hollow polygons.
index clamping	Where possible, OpenGL treats color and stencil indexes as bitfields rather than numbers. Thus indexes are masked, rather than clamped, to the supported range of the framebuffer.
input and output	I/O in OpenGL is usually handled by X calls. See Chapter 4 for more information.
integer colors	Signed integer color components (red, green, blue, or alpha) are linearly mapped to floating point such that the most negative integer maps to -1.0 and the most positive integer maps to 1.0. This mapping occurs when the color is specified, before it replaces the current color. Unsigned integer color components are linearly mapped to floating point such that 0 maps to 0.0 and the largest representable integer maps to 1.0. This mapping occurs when the color is specified, before it replaces the current color.
integer normals	Integer normal components are mapped just like signed color components, such that the most negative integer maps to -1.0, and the most positive integer maps to 1.0.

invariance	OpenGL guarantees certain invariances that IRIS GL does not. For example, OpenGL guarantees that identical code sequences sent to the same system, differing only in the blending function specified, will generate the same pixel fragments. (The fragments may be different if blending is enabled and disabled, however.)
lighting equation	The OpenGL lighting equation differs slightly from the IRIS GL equation. OpenGL supports separate attenuation for each light source, rather than a single attenuation for all the light sources as in IRIS GL, and OpenGL regularizes the equation so that ambient, diffuse, and specular lighting contributions are all attenuated. In addition, OpenGL lets you specify separate colors for the ambient, diffuse, and specular intensities of light sources, and for the ambient, diffuse, and specular reflectance of materials. All OpenGL light and material colors must include an alpha value, though only the diffuse material-color alpha value is actually used for lighting. Setting the specular exponent to zero does *not* defeat specular lighting in OpenGL. OpenGL supports local lights in color index mode. IRIS GL does not.
line stipple repeat	OpenGL line stipple repeat is clamped to [1, 256], while IRIS GL clamps this value to [1, 255].
mapw()	OpenGL utilities don't directly support mapping between object and window coordinates. If you specify the right projection matrix and viewport, you may be able to achieve the same effect using **gluProject()**.
material color	In IRIS GL, you could call **lmcolor()** between a call to **bgnprimitive()** and the corresponding **endprimitive()** call. In OpenGL, you can't call **glColorMaterial()** between a **glBegin()** and its corresponding **glEnd()**. Material coloring in IRIS GL it was connected with lighting models. In OpenGL, it's part of the OpenGL state.

matrix mode	All OpenGL matrix operations operate on the current matrix, rather than on a particular matrix, as do the IRIS GL **ortho()**, **ortho2()**, **perspective()**, and **window()** commands. All OpenGL matrix operations except **glLoadIdentity()** and **glLoadMatrix()** multiply the current matrix rather than replacing it (as do **ortho()**, **ortho2()**, **perspective()**, and **window()** in IRIS GL).
mipmaps, automatic generation	The OpenGL texture interface does not support automatic generation of mipmap images. GLU does support automatic generation of mipmap images for both 1D and 2D textures; however, GLU mipmap generation isn't as flexible as that of IRIS GL. (For instance, GLU doesn't currently allow you to set weights for the texels when you average texels to generate a small mipmap from a larger one.)
mixed-model	For an extensive discussion of this topic, see Chapter 4, "OpenGL in the X Window System." Note in particular that IRIS GL mixed-model routines had, in some cases, names confusingly similar to unrelated OpenGL routines; see "Function Naming Conventions" in Chapter 4 for details.
move/draw/pmove/pdraw/ pclos	OpenGL supports only glBegin/glEnd style graphics, because it does not maintain a current graphics position. The scalar parameter specification of the old move/draw commands is accepted by OpenGL for all vertex related commands, however.
mprojection mode	IRIS GL did not transform geometry by the modelview matrix while in projection matrix mode. OpenGL always transforms by both the modelview nad the projection matrix, regardless of matrix mode.
MSINGLE mode	See the entry for "single matrix mode" in this appendix.
multi-buffer drawing	OpenGL renders to each color buffer individually, rather than computing a single new color value based on the contents of one color buffer and writing it to all the enabled color buffers, as IRIS GL did.
multisampling	Multisampling is supported only in an extension to OpenGL. See the *OpenGL on Silicon Graphics Systems* document.

normals	When OpenGL transforms normals, it uses the exact inverse of the modelview matrix. Thus, all scale commands, even scale commands with the same scale values for x, y, and z, affect the lengths of transformed normals. Avoid calling **glScale()** if you want the performance advantage of leaving GL_NORMALIZE disabled.
NURBS	OpenGL supports NURBS with a combination of core capability (evaluators) and GLU support. GLU currently supports only Bernstein polynomials, not all splines; in the future, GLU may support changing the basis matrix to handle all splines. See the gluNewNurbsRenderer reference page.
old polygon mode	Aliased OpenGL polygons are always point sampled. The old polygon compatibility mode of the IRIS GL, where pixels outside the polygon perimeter were included in its rasterization, is not supported. If your code uses old polygon mode, it's probably for rectangles. Old polygon mode rectangles appear one pixel wider and higher.
packed color formats	OpenGL accepts colors as 8-bit components, but these components are treated as an array of bytes rather than as bytes packed into larger words. By encouraging array indexing rather than shifting, OpenGL promotes endian-invariant programming. Just as IRIS GL accepted packed colors both for geometric and pixel rendering, OpenGL accepts arrays of color components for geometric and pixel rendering
patches	OpenGL does not support IRIS GL patches. Use of evaluators is recommended.
per-bit color writemask	OpenGL writemasks for color components enable or disable changes to the entire component (red, green, blue, or alpha), not to individual bits of components. Note that per-bit writemasks are supported for both color indexes and stencil indexes, however.
per-bit depth writemask	OpenGL writemasks for depth components enable or disable changes to the entire component, not to individual bits of the depth component.

performance	The performance of an OpenGL program depends in part on whether certain OpenGL features are used. A straightforward port of an IRIS GL program will probably require tuning to achieve maximum performance in OpenGL. For some tips on maximizing OpenGL performance, see “Performance” on page 14.
pick	The OpenGL utility library includes support for generating a pick matrix. See the gluPickMatrix reference page.
pixel coordinates	OpenGL and IRIS GL agree that the origin of a window’s coordinate system is at its lower left corner. OpenGL places the origin at the lower left corner of this pixel, however, while IRIS GL placed it at the center of the lower left pixel. Note that the X Window System assumes an upper left corner for its origin.
pixel fragments	Pixels drawn by **glDrawPixels()** or **glCopyPixels()** are always rasterized and converted to fragments. The resulting fragments are textured, fogged, depth buffered, blended, and so on, just as if they had been generated from geometric points. Fragment data that are not provided by the source pixels are augmented from the current raster position. For example, RGBA pixels take the raster position Z and texture coordinates. Depth pixels take the raster position color and texture coordinates.
pixel zoom	OpenGL negative zoom factors reflect about the current graphics position. IRIS GL doesn’t define the operation of negative zoom factors, and instead provides RIGHT_TO_LEFT and TOP_TO_BOTTOM reflection pixmodes. These reflection modes reflect in place, rather than about the current raster position. OpenGL doesn’t define reflection modes. Also, OpenGL allows fractional zoom factors.
pixmode	OpenGL pixel transfers operate on individual color components, rather than on packed groups of four 8-bit components as does IRIS GL. While OpenGL provides substantially more pixel capability than IRIS GL, it does not support packed color constructs, and it does not allow color components to be reassigned (red to green, red to blue, and so on) during pixel copy operations.

polf()/poly()	OpenGL provides no direct support for vertex lists other than display lists. Functions like **polf()** and **poly()** can easily be implemented using the OpenGL API, however.
polygon mode	OpenGL supports only filled, outlined, and dotted polygons. There is no hollow polygon mode as in IRIS GL. OpenGL polygon modes are specified separately for front and back facing polygons, while IRIS GL shares a single mode for all polygons.
polygon provoking vertex	Flat shaded IRIS GL polygons took the color of the last vertex specified, while OpenGL polygons take the color of the first vertex specified. (Note that this is true only for the GL_POLYGON primitive, not for triangles, triangle strips, and other primitive types, each of which take their colors from different vertices. See the reference page for glShadeModel for details.)
polygon stipple	In IRIS GL the polygon stipple pattern is screen-relative. In OpenGL it is window-relative.
polygon vertex count	There is no limit to the number of vertexes between **glBegin()** and **glEnd()** in OpenGL, even for **glBegin(POLYGON)**. In IRIS GL polygons are limited to no more than 255 vertexes.
readdisplay	Reading pixels outside window boundaries is properly a window system capability, rather than a renderer capability. Silicon Graphics supports an extension to X that replaces the IRIS GL **readdisplay()** command. See the XReadDisplay reference page.
RealityEngine graphics features	Many of the special RealityEngine graphics features of IRIS GL (including multisampling and some texture-mapping features) have been implemented as extensions to OpenGL.
relativemove/draw/pmove/ pdraw/pclos	OpenGL does not maintain a current graphics position, and therefore is unable to support relative vertex operations. The semantics of such operations can easily be emulated by using the matrix and **glTranslate()**.
reset linestyle	IRIS GL **resetls()** has not been supported for some time, and is not supported by OpenGL.

RGBA logicop()	OpenGL 1.0 does not support logical operations on RGBA buffers. OpenGL 1.1 adds support for logical operations on RGBA buffers.
sbox()	**sbox()** is an IRIS GL rectangle primitive that is well defined only if transformed without rotation, and is designed to be faster than standard rectangles. While OpenGL does not support such a primitive, it can be tuned to render rectangles very quickly when the matrixes and other modes are in states that simplify calculations.
scalar arguments	All OpenGL commands that are accepted between **glBegin()** and **glEnd()** have entry points that accept scalar arguments. For example, `glColor4f(red,green,blue,alpha)`.
scissor	OpenGL **glScissor()** does not track the viewport. The IRIS GL **viewport()** command automatically updates the scrmask.
scrbox()	OpenGL doesn't support bounding box computation.
scrsubdivide()	OpenGL doesn't support explicit screen subdivision. **scrsubdivide()** was used in IRIS GL to handle perspective properly when interpolating colors and textures. Most Silicon Graphics platforms now handle texture interpolation correctly, but not all platforms do perspective-corrected color interpolation. If you notice a perspective problem in interpolation, try specifying this hint: `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)` Under some circumstances, that may improve the interpolation. GL_NICEST specifies quality at the expense of speed, however, so if speed is a high priority you may be forced to settle for linear interpolation.
single matrix mode	OpenGL always maintains two matrices: the modelview matrix and the projection matrix. While an OpenGL implementation may consolidate these into a single matrix for performance reasons, it must always present the two-matrix model to the programmer. See "Porting MSINGLE Mode Code" in Chapter 3 for more information.
specular exponent, setting to zero	See the entry for "lighting equation" in this appendix.

stencil	When there is no depth buffer, or when the depth buffer is not enabled, the **glStencilOp()** argument *zpass* controls stencil operation when the stencil test passes. The IIS GL stencil operation is defined by its *pass* parameter (equivalent to OpenGL *zfail*) in this case.
stereo	Stereo rendering on RealityEngine graphics systems under OpenGL is accomplished by choosing an appropriate X visual.
subpixel mode	All OpenGL rendering is subpixel positioned—subpixel mode is always on.
swapbuffers()	Anything involving multiple buffers or windowing must be done in X or with GLUT.
swaptmesh()	OpenGL does not support the **swaptmesh()** capability. It does offer two types of triangle meshes, however: one that corresponds to the default "strip" behavior of the IRIS GL, and another that corresponds to calling **swaptmesh()** prior to the third and all subsequent vertexes when using IRIS GL.
texture filtering	OpenGL textures are filtered with a border when they are clamped. IRIS GL does not use the border data in this case.
texture lookup tables	Texture lookup tables aren't supported in OpenGL 1.0 but are supported in OpenGL 1.1.
texture scaling, automatic	The OpenGL texture interface does not support automatic scaling of images to power-of-two dimensions. However, the GLU supports image scaling.
texturing, 3D	Three-dimensional texturing is provided as part of an extension to OpenGL.
uniform scaling	If you use only unit-length normals in IRIS GL, and if the modelview matrix is the product only of rotations and uniform scales, you don't need to enable normalization of the normal vectors. In OpenGL, however, uniform scaling does affect the length of normal vectors, even unit-length normals
vector arguments	All OpenGL commands that are accepted between **glBegin()** and **glEnd()** have entry points that accept vector arguments. For example, **glColor4fv(v)**.

window management	OpenGL includes no window system commands. It is always supported as an extension to a window or operating system that includes capability for device and window control. Each extension provides a system-specific mechanism for creating, destroying, and manipulating OpenGL rendering contexts. For example, the OpenGL extension to the X window system (GLX) includes roughly ten commands for this purpose. IRIS GL commands such as **gconfig()** and **drawmode()** are not implemented by OpenGL. In OpenGL, windows have static frame buffer configurations.
window offset	IRIS GL returned viewport and character position in screen, rather than window, coordinates. OpenGL always deals with window coordinates.
z-buffer, reading from	If you wanted to read from the *z*-buffer in IRIS GL, you specified that buffer with **readsource()** and then used **lrectread()** or **rectread()** to do the reading. If you want to read from the *z*-buffer in OpenGL, you simply specify that buffer as a parameter to **glReadPixels()**.
z-buffer sizing	Changing the depth of the *z*-buffer can be done by selecting an appropriate visual.
z rendering	OpenGL does not support rendering colors to the depth buffer. It does allow for additional color buffers, which can be implemented using the same memory that is used for depth buffers in other window configurations—but these additional color buffers cannot share memory with the depth buffer in any single configuration.

Appendix C

OpenGL Names, Types, and Error

This appendix provides some background information on OpenGL command names, defined types, and error handling. It's intended mainly as a quick reference. For more detailed information, refer to the *OpenGL Programming Guide*.

OpenGL Command Names

This section describes the naming convention for OpenGL calls. For a complete list of OpenGL equivalents to IRIS GL routines, see Appendix A, "OpenGL Commands and Their IRIS GL Equivalents."

- **gl—**OpenGL commands begin with the gl prefix (**glEnable()**, **glTranslatef()**, and so on).
- **glu—**OpenGL Utility Library (GLU) commands begin with the glu prefix (**gluDisk()**, **gluErrorString()**, and so on).
- **glX—**Commands that belong to the OpenGL extension to X (GLX) begin with the glX prefix (**glXChooseVisual()**, **glXCopyContext()**, and so on).
- **glut**—Commands in the GLUT (OpenGL Utility Toolkit) library begin with a glut prefix (**glutInit**, **glutMenuStatusFunc**, and so on).

OpenGL commands are formed by a root name, optionally followed by up to four characters. The first character indicates the number of arguments. The second character, or pair of characters, specifies the type of the arguments. Table C-1 lists the character suffixes and the corresponding argument types (some of these values might be different on a 64-bit architecture).

Table C-1 Command Suffixes and Corresponding Argument Types

Letter	Type	C Type
b	8-bit integer	char
s	16-bit integer	short

Table C-1 (continued) Command Suffixes and Corresponding Argument Types

Letter	Type	C Type
i	32-bit integer	long
f	32-bit floating point	float
d	64-bit floating point	double
ub	8-bit unsigned integer	unsigned char
us	16-bit unsigned integer	unsigned short
ui	32-bit unsigned integer	unsigned long

The final character, if present, is **v**. The **v** indicates that the command takes a pointer to an array of values—a vector—rather than a series of individual arguments.

IRIS GL used a similar mechanism for some commands. The predecessor to **glVertex*()**, for example, was **v()**, which also used a suffix to specify the number and type of its arguments.

Here are some examples of OpenGL command naming:

```
void glVertex2i( GLint x, GLint y);
void glVertex3f( GLfloat x, GLfloat y, GLfloat z);
void glVertex3dv( const GLdouble *v) ;
```

The first version of the vertex call, **glVertex2i()**, takes two integer arguments. The second, **glVertex3f()**, is a three-dimensional version, which expects three floats. The third version, **glVertex3dv(),** expects an argument in the form of a vector, which is a pointer to an array. In this case, the array should have three elements.

OpenGL Defined Types

Table C-2 lists C data types and their equivalent OpenGL defined types.

Table C-2 OpenGL Equivalents to C Data Types

C Type	Equivalent OpenGL Type
bitmask value	GLbitfield
boolean value	GLboolean
double	GLdouble
double value clamped to [0.0, 1.0]	GLclampd
enumerated type	GLenum
float	GLfloat
float value clamped to [0.0, 1.0]	GLclampf
long	GLint
short	GLshort
signed char	GLbyte
unsigned char	GLubyte
unsigned int	GLuint
unsigned short	GLushort
void	GLvoid

Error Handling

When an error occurs, OpenGL sets an error flag to the appropriate error value. You can test error conditions using the **glGetError()** call, which returns the error number. Table C-3 lists possible error values. For details, see the reference page for **glGetError()**.

Table C-3 **glGetError()** Return Values

Error	Description	Command Ignored?
NO_ERROR	No error	No
INVALID_ENUM	Enumerated argument out of range	Yes
INVALID_VALUE	Numeric argument out of range	Yes
INVALID_OPERATION	Operation illegal in current state	Yes
STACK_OVERFLOW	Command would cause a stack overflow	Yes
STACK_UNDERFLOW	Command would cause a stack underflow	Yes
OUT_OF_MEMORY	Not enough memory left to execute command	Unknown

Example OpenGL Program With the GLUT Library

This program uses OpenGL and the GLUT library to display a planet rotating around the sun. It demonstrates how to composite modeling transformations to draw translated and rotated models. Pressing the left, right, up, and down arrow keys alters the rotation of the planet around the sun.

```
/*
 *  planet.c
 *  This program shows how to composite modeling transformations
 *  to draw translated and rotated models.
 *  Interaction:  pressing the d and y keys (day and year)
 *  alters the rotation of the planet around the sun.
 */
#include <GL/glut.h>
#include <stdlib.h>

static int year = 0, day = 0;

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);

   glPushMatrix();
   glutWireSphere(1.0, 20, 16);   /* draw sun */
   glRotatef((GLfloat)year, 0.0, 1.0, 0.0);
   glTranslatef(2.0, 0.0, 0.0);
   glRotatef((GLfloat)day, 0.0, 1.0, 0.0);
   glutWireSphere(0.2, 10, 8);    /* draw smaller planet */
   glPopMatrix();
   glutSwapBuffers();
}
```

```
void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (GLfloat)w/(GLfloat)h, 1.0, 20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}

/* ARGSUSED1 */
void keyboard (unsigned char key, int x, int y)
{
   switch (key) {
      case 'd':
         day = (day + 10) % 360;
         glutPostRedisplay();
         break;
      case 'D':
         day = (day - 10) % 360;
         glutPostRedisplay();
         break;
      case 'y':
         year = (year + 5) % 360;
         glutPostRedisplay();
         break;
      case 'Y':
         year = (year - 5) % 360;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
      default:
         break;
   }
}

int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
   glutInitWindowSize(500, 500);
```

```
    glutInitWindowPosition(100, 100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Appendix E

Example Program Using Xt and a WorkProc

This appendix contains an example program that uses Xt, IRIS IM, and the IRIS IM version of the Silicon Graphics widget. The program displays a planet with a moon, orbiting a sun, and uses a WorkProc for the animation.

```
/* opensolar.c
 * opensolar displays a planet with a moon, orbiting a sun.
 * To compile:
 * cc -O -o opensolar opensolar.c -lXm -lGLw -lm -lGLU -lGL
 */

#include <Xm/Xm.h>
#include <Xm/Frame.h>
#include <Xm/Form.h>
#include <X11/keysym.h>
#include <X11/StringDefs.h>
#include <GL/GLwMDrawA.h>

#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glx.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "malloc.h"

typedef struct _spin {
  short year;
} SPINDATA, *SPINPTR;

/* function prototypes */
void main(int argc, char **argv);
void initCB (Widget w, XtPointer client_data,
             XtPointer call_data);
void exposeCB (Widget w, XtPointer spin,
               XtPointer call_data);
void resizeCB (Widget w, XtPointer spin,
               XtPointer call_data);
```

```
void inputCB (Widget w, XtPointer client_data,
              XtPointer call_data);
Boolean drawWP (XtPointer spin);
void drawscene(SPINPTR spin);
void setbeachball(int stripes);
void beachball(unsigned long color1, unsigned long color2);

XtAppContext app_context;
XtWorkProcId workprocid = NULL;

GLXContext glx_context;
Display * global_display;
Window global_window;


/* main
 * This program shows a solar system, with a sun, planet, and
 * moon (in OpenGL). The user can exit with the ESCape key
 * or through the window manager menu.
 */
void main(int argc, char **argv)
{
  Arg wargs[15];
  int n;
  Widget glw, toplevel, frame, form;
  SPINPTR spin;
  static String fallback_resources[] = {
    "*frame*shadowType: SHADOW_IN", "*glwidget*width: 750",
    "*glwidget*height: 600", "*glwidget*rgba: TRUE",
    "*glwidget*doublebuffer: TRUE",
    "*glwidget*allocateBackground: TRUE", NULL
  };

  /* create main data structure, spin pointer */
  spin = (SPINPTR) malloc (sizeof (SPINDATA));
  spin->year = 0;
  toplevel = XtAppInitialize(
     &app_context, /* Application context */
     "Opensolar", /* Application class */
     NULL, 0, /* command line option list */
     &argc, argv, /* command line args */
     fallback_resources,
     NULL, /* argument list */
     0); /* number of arguments */
```

```
  n = 0;
  form = XmCreateForm(toplevel, "form", wargs, n);
  XtManageChild(form);

  n = 0;
  XtSetArg(wargs[n], XtNx, 30);
  n++;
  XtSetArg(wargs[n], XtNy, 30);
  n++;
  XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM);
  n++;
  XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM);
  n++;
  XtSetArg(wargs[n], XmNrightAttachment, XmATTACH_FORM);
  n++;
  XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM);
  n++;
  XtSetArg(wargs[n], XmNleftOffset, 30);
  n++;
  XtSetArg(wargs[n], XmNbottomOffset, 30);
  n++;
  XtSetArg(wargs[n], XmNrightOffset, 30);
  n++;
  XtSetArg(wargs[n], XmNtopOffset, 30);
  n++;
  frame = XmCreateFrame (form, "frame", wargs, n);
  XtManageChild (frame);

  n = 0;
  glw = GLwCreateMDrawingArea(frame, "glwidget", wargs, n);
  XtManageChild (glw);
  XtAddCallback(glw, GLwNginitCallback, initCB,
                (XtPointer) NULL);
  XtAddCallback(glw, GLwNexposeCallback, exposeCB,
                (XtPointer) spin);
  XtAddCallback(glw, GLwNresizeCallback, resizeCB,
                (XtPointer) spin);
  XtAddCallback(glw, GLwNinputCallback, inputCB,
                (XtPointer) NULL);

  XtRealizeWidget(toplevel); /* instantiate it now */
  XtAppMainLoop(app_context); /* loop for events */
} /* end main() */
```

```
/* initCB
 * The initCB subroutine initializes graphics modes and
 * transformation matrices.
 */
void initCB (Widget w, XtPointer client_data,
             XtPointer call_data)
{
  Arg args[1];
  XVisualInfo *vi;

  XtSetArg(args[0], GLwNvisualInfo, &vi);
  XtGetValues(w, args, 1);

  global_display = XtDisplay(w);
  global_window = XtWindow(w);
  glx_context = glXCreateContext(global_display, vi, 0,
                                 GL_FALSE);
} /* end initCB() */


/* exposeCB() and resizeCB() are called when the window
 * is uncovered, moved, or resized.
 */
void exposeCB (Widget w, XtPointer ptr, XtPointer call_data)
{
  SPINPTR spin;
  static char firstTime = 0x1;
  GLwDrawingAreaCallbackStruct *call_ptr;

  call_ptr = (GLwDrawingAreaCallbackStruct *) call_data;
  GLwDrawingAreaMakeCurrent(w, glx_context);
  if (firstTime) {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(45.0, (GLfloat)(call_ptr->width)
                   /(GLfloat)(call_ptr->height), 1.0, 25.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity ();
    glTranslatef(0.0, 0.0, -12.0);
    workprocid = XtAppAddWorkProc(app_context, drawWP, ptr);
      /* ptr is spin */
    firstTime = 0;
```

```
  }
  spin = (SPINPTR) ptr;
  drawscene(spin);
}

void resizeCB (Widget w, XtPointer ptr, XtPointer call_data)
{
  GLwDrawingAreaCallbackStruct *call_ptr;
  SPINPTR spin;

  spin = (SPINPTR) ptr;
  call_ptr = (GLwDrawingAreaCallbackStruct *) call_data;
  GLwDrawingAreaMakeCurrent(w, glx_context);
  glViewport (0, 0, (GLsizei) (call_ptr->width-1),
              (GLsizei) (call_ptr->height-1));
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity ();
  gluPerspective(45.0, (GLfloat)(call_ptr->width) /
                 (GLfloat)(call_ptr->height), 1.0, 25.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity ();
  glTranslatef(0.0, 0.0, -12.0);
  drawscene(spin);
}


/* inputCB() handles all types of input from the GL widget.
 * The KeyRelease handles the ESCape key, so that it exits
 * the program.
 */
void inputCB (Widget w, XtPointer client_data,
              XtPointer call_data)
{
  char buffer[1];
  KeySym keysym;
  GLwDrawingAreaCallbackStruct *call_ptr;
  XKeyEvent *kevent;

  call_ptr = (GLwDrawingAreaCallbackStruct *) call_data;
  kevent = (XKeyEvent *) (call_ptr->event);
  switch(call_ptr->event->type) {
  case KeyRelease:
    /* Must convert the keycode to a keysym before
     * checking if it is an escape
     */
```

```
    if (XLookupString(kevent,buffer,1,&keysym,NULL) == 1
                      && keysym == (KeySym)XK_Escape)
      exit(0);
    break;
  default:
    break;
  }
}


/* drawWP() is called by the WorkProc. When the scene
 * is in automatic motion, the WorkProc calls this routine,
 * which adds 1 degree (10 tenths) to the cumulative amount
 * of rotation. drawscene() is called, so the image is
 * redrawn. It returns(FALSE) so the WorkProc does not
 * discontinue operation.
 */
Boolean drawWP (XtPointer ptr)
{
  SPINPTR spin;

  spin = (SPINPTR) ptr;
  spin->year = (spin->year + 10) % 3600;
  drawscene (spin);
  return (FALSE);
}

/* drawscene
 * drawscene calculates angles relative to the spin->year
 * and then draws sun, planet, and moon.
 */
void drawscene(SPINPTR spin)
{
  short sunangle;
  /* actual dist is 1.5e8 km; mult by 3.0e-8 fudgefactor */
  float earthdist = 4.5;
  short dayangle;
  float earthscale = 0.5;
  short monthangle;
  float moondist = 0.9;
  float moonscale = 0.2;

  glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT);

  glPushMatrix();
```

```
  glRotatef(10.0, 1.0, 0.0, 0.0); /* tilt entire scene */
  glPushMatrix();
  sunangle = (spin->year*365/25) % 3600;
  /* sun rotates on axis every 25 days */
  glRotatef(.1*(sunangle), 0.0, 1.0, 0.0);
  /* cpack format color1, color2 */
  /* swapped by hand: was beachball(0x20C0FF, 0x200FFFF); */
  beachball(0xFFC02000, 0xFFFF0020);
  glPopMatrix();
  glPushMatrix();
  glRotatef(.1*(spin->year), 0.0, 1.0, 0.0);
  glTranslatef(earthdist, 0.0, 0.0);
  glPushMatrix();
  dayangle = (spin->year*50) % 3600;
  /* dayangle fudged so earth rotation can be seen */
  glRotatef(.1*(dayangle), 0.0, 1.0, 0.0);
  glScalef(earthscale, earthscale, earthscale);
  glColor3f(0.0, 0.0, 1.0);
  /* swap by hand; was beachball(0xFF0000, 0xC02000);*/
  beachball(0x0000FF00, 0x0020C000); /* earth */
  glPopMatrix();
  monthangle = (spin->year*365/28) % 3600;
  glRotatef(.1*(monthangle), 0.0, 1.0, 0.0);
  glTranslatef(moondist, 0.0, 0.0);
  glScalef(moonscale, moonscale, moonscale);
  glColor3f(1.0, 1.0, 1.0);
  /* swap by hand; was beachball(0xFFFFFF, 0xC0C0C0); */
  beachball(0xFFFFFF00, 0xC0C0C000); /* moon */
  glPopMatrix();
  glPopMatrix();
  glXSwapBuffers(global_display, global_window);
} /* end drawscene() */


/*
 * BEACHBALL
 */

/* three dimensional vector */
typedef float vector[3];
vector front =  { 0.0, 0.0, 1.0 };
vector back =   { 0.0, 0.0, -1.0 };
vector top =    { 0.0, 1.0, 0.0 };
vector bottom = { 0.0, -1.0, 0.0 };
vector right =  { 1.0, 0.0, 0.0 };
```

```
vector left =   { -1.0, 0.0, 0.0 };
vector center = { 0.0, 0.0, 0.0 };

/* Number of colored stripes. Should be even to look right */
#define BEACHBALL_STRIPES 12
/* Default number of polygons making up a stripe. Should */
/* be even */
#define BEACHBALL_POLYS 16

/* array of vertices making up a stripe */
vector stripe_point[BEACHBALL_POLYS + 3];

/* has the beachball been initialized */
Boolean beachball_initialized = FALSE;

/* Number of polygons making up a stripe */
int beachball_stripes;

/* Number of vertices making up a stripe */
int stripe_vertices;

/* Initializes beachball_point array to a stripe of unit */
/* radius. */
void setbeachball(int stripes)
{
  int i,j;
  float x,y,z; /* vertex points */
  float theta,delta_theta; /* angle from top pole to bottom*/
  float offset; /* offset from center of stripe to vertex */
  /* radius of cross-section at current latitude */
  float cross_radius;
  float cross_theta; /* angle occupied by a stripe */

  beachball_stripes = stripes;

  /* polys distributed by even angles from top to bottom */
  delta_theta = M_PI/((float)BEACHBALL_POLYS/2.0);
  theta = delta_theta;
  cross_theta = 2.0*M_PI/(float)beachball_stripes;

  j = 0;
  stripe_point[j][0] = top[0];
  stripe_point[j][1] = top[1];
  stripe_point[j][2] = top[2];
  j++;
```

```
  for (i = 0; i < BEACHBALL_POLYS; i += 2) {
    cross_radius = fsin(theta);
    offset = cross_radius * ftan(cross_theta/2.0);

    stripe_point[j][0] = - offset;
    stripe_point[j][1] = fcos(theta);
    stripe_point[j][2] = cross_radius;
    j++;

    stripe_point[j][0] = offset;
    stripe_point[j][1] = stripe_point[j-1][1];
    stripe_point[j][2] = stripe_point[j-1][2];
    j++;

    theta += delta_theta;
  } /* end for */

  stripe_point[j][0] = bottom[0];
  stripe_point[j][1] = bottom[1];
  stripe_point[j][2] = bottom[2];

  stripe_vertices = j + 1;

  beachball_initialized = TRUE;
}


/* Draws a canonical beachball. The colors are cpack values
 * when in RGBmode.
 */
void beachball(unsigned long c1, unsigned long c2)
{
  float angle, delta_angle;
  int i, j;

  if (! beachball_initialized)
    setbeachball(BEACHBALL_STRIPES);

  angle = 0.0;
  delta_angle = 360.0/(float)beachball_stripes;

  for (i = 0; i < beachball_stripes; i++) {
    if ( i%2 == 0)
      glColor4ubv((GLubyte *)(&c1));
```

```
    else
      glColor4ubv((GLubyte *)(&c2));
    glPushMatrix();
    glRotatef(angle, 0.0, 1.0, 0.0);
    angle += delta_angle;

    glBegin(GL_TRIANGLE_STRIP);
    for (j = 0; j < stripe_vertices; j++)
      glVertex3fv(stripe_point[j]);
    glEnd();
    glPopMatrix();
  }
}
```

Appendix F

Example Mixed-Model Programs With Xlib

This appendix contains two example mixed-model programs that use Xlib. Each example program is shown first in IRIS GL, then in OpenGL.

Example One: iobounce.c

iobounce.c is a simple interactive program that bounces a ball around a 2D surface. Users can use the mouse buttons to change the velocity of the ball. The IRIS GL version of the program is presented first, then the OpenGL version.

IRIS GL Version of iobounce.c

The IRIS GL version of *iobounce.c* is a "pure" IRIS GL program; it does not contain X calls.

```
/*              iobounce.c:
 *              "pool" ball that "bounces" around a 2-d "surface".
 *              RIGHTMOUSE stops ball
 *              MIDDLEMOUSE increases y velocity
 *              LEFTMOUSE increases x velocity
 */

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

long xmaxscrn, ymaxscrn;  /* maximum size of screen in x and y */

#define XMIN 100
#define YMIN 100
#define XMAX 900
#define YMAX 700

long xvelocity = 0, yvelocity = 0;
```

```
main()
{
    Device dev;
    short val;
    long sizex, sizey;

    initialize();

    while (TRUE) {
       while (qtest()) {
          dev = qread(&val);
          switch (dev) {
                case REDRAW:  /* redraw window re: move/resize/push/pop */
                    reshapeviewport();
                    ortho2(XMIN - 0.5, XMAX + 0.5, YMIN - 0.5,
                           YMAX + 0.5);
                    drawball();
                    break;
                case LEFTMOUSE:     /* increase xvelocity */
                    if (xvelocity >= 0)
                        xvelocity++;
                    else
                        xvelocity--;
                    break;
                case MIDDLEMOUSE:   /* increase yvelocity */
                    if (yvelocity >= 0)
                        yvelocity++;
                    else
                        yvelocity--;
                    break;
                case RIGHTMOUSE:     /* stop ball */
                    xvelocity = yvelocity = 0;
                    break;
                case ESCKEY:
                    gexit();
                    exit(0);
             }
          }
          drawball();
    }
}

initialize() {

    xmaxscrn = getgdesc(GD_XPMAX)-1;
```

```
    ymaxscrn = getgdesc(GD_YPMAX)-1;
    prefposition(xmaxscrn/4,xmaxscrn*3/4,ymaxscrn/4,ymaxscrn*3/4);
    winopen("iobounce");
    winconstraints();

    doublebuffer();
    gconfig();
    shademodel(FLAT);

    ortho2(XMIN - 0.5, XMAX + 0.5, YMIN - 0.5, YMAX + 0.5);

    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MIDDLEMOUSE);
    qdevice(RIGHTMOUSE);
}

drawball() {
    static xpos = 500,ypos = 500;
    long radius = 10;

    color(BLACK);
    clear();
    xpos += xvelocity;
    ypos += yvelocity;
    if (xpos > XMAX - radius ||
        xpos < XMIN + radius) {
        xpos -= xvelocity;
        xvelocity = -xvelocity;
    }
    if (ypos > YMAX - radius ||
        ypos < YMIN + radius) {
        ypos -= yvelocity;
        yvelocity = -yvelocity;
    }
    color(YELLOW);
    circfi(xpos, ypos, radius);
    swapbuffers();
}
```

OpenGL Version of iobounce.c

Thie example contains the OpenGL version of *iobounce.c*. Windowing and event handling are now controlled with Xlib, rather than IRIS GL calls.

```
/*                  iobounce.c:
 *
 *   "pool ball" that "bounces" around a 2-d "surface".
 *            RIGHTMOUSE stops ball
 *            MIDDLEMOUSE increases y velocity
 *            LEFTMOUSE increases x velocity
 *
 *           (ported from ~4Dgifts/example/grafix/iobounce.c)
 */

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

#define XMIN 100
#define YMIN 100
#define XMAX 900
#define YMAX 700

#define BLACK           0
#define YELLOW          3

#define LEFTMOUSE       3
#define MIDDLEMOUSE     2
#define RIGHTMOUSE      1

#define TRUE            1
#define FALSE           0

long xmaxscrn, ymaxscrn; /* maximum size of screen /*
                         /* in x and y */

Display *dpy;            /* The X server connection */
Atom del_atom;           /* WM_DELETE_WINDOW atom   */
Window glwin;            /* handle to the GL window */
```

```
XEvent event;

static void openwindow(char *);
static void drawball(void);
static void clean_exit(void);

long xvelocity = 0, yvelocity = 0;

main(int argc, char *argv[])
{
    int myExpose, myConfigure,
        myButtRelease, myKeyPress,
        myButtonNumber;   /* store which events occur */
    long xsize, ysize;



    myExpose = myConfigure = myButtRelease = myKeyPress =
               myButtonNumber = FALSE;

    openwindow(argv[0]);

    while (TRUE) {

        KeySym keysym;
        char buf[4];

    /* this "do while" loop does the 'get events' half of the     */
    /* "get events,process events" action of the infinite while.  */
    /* This is to ensure the event queue is always drained before */
    /* the events that have come in are processed.                */
    while (XEventsQueued(dpy,QueuedAfterReading)) { /* end "do { } while"
                      * XEventsQueued(dpy,QueuedAfterReading)
                      * is like qtest()--it only tells you if
                      * there're any events presently in the
                      * queue.it does not disturb the event
                      * queue's contents in any way.       */

            XNextEvent(dpy, &event);
            switch (event.type) {

            /* "Expose" events are sort of like "REDRAW" in gl-speak in
             * terms of when a window or a previously invisible part
             * becomes visible */
                case Expose:   /* Exposures */
```

```
        myExpose = TRUE;
        break;

/* "ConfigNotify" events are like "REDRAW" in terms of changes to
 * a window's size or position.*/
    case ConfigureNotify:  /* Resize GL manually */
        xsize = event.xconfigure.width;
        ysize = event.xconfigure.height;
        myConfigure = TRUE;
        break;

/* Wait for "ButtonRelease" events so the queue doesn't fill up
 * the way it would if the user sits on ButtonPresss. */
    case ButtonRelease:
        if (event.xbutton.button == Button1) {
            myButtonNumber = LEFTMOUSE;
            myButtRelease = TRUE;
        } else if (event.xbutton.button ==
                   Button2) {
            myButtonNumber = MIDDLEMOUSE;
            myButtRelease = TRUE;
        } else if (event.xbutton.button ==
                   Button3) {
            myButtonNumber = RIGHTMOUSE;
            myButtRelease = TRUE;
        }  /* twirl the green sphere */
        break;

/* "ClientMessage" is generated if the WM itself is dying
 * and sends an exit signal to any running prog. */
    case ClientMessage:
        if (event.xclient.data.l[0] == del_atom)
            clean_exit();
        break;

/* "KeyPress" events are those that would be generated before
 * whenever queueing up any KEYBD key via qdevice. */
    case KeyPress:
       /* save out which unmodified key (i.e. the key was not
        * modified w/something like "Shift", "Ctrl", or "Alt")
        * got pressed for use below. */
        XLookupString((XKeyEvent *)&event, buf, 4, &keysym, 0);
        myKeyPress = TRUE;
        break;
```

```
        }  /* end switch (event.type) */
    }

/* On an "Expose" event, redraw the affected pop'd or
 * de-iconized window
 */
    if (myExpose) {
        drawball();       /* draw the GL stuff */
        myExpose = FALSE; /* reset flag--queue now empty */
    }

/* On a "ConfigureNotify" event, the GL window has either
 * been moved or resized.  Respond accordingly and then
 * redraw its contents.
 */

    if (myConfigure) {
        glViewport(0, 0, xsize, ysize);
        glLoadIdentity();
        gluOrtho2D(XMIN-0.5, XMAX+0.5, YMIN-0.5, YMAX+0.5);
        drawball();           /* draw the GL stuff */
        myConfigure = FALSE;  /* reset flag--queue now
                               * empty */
    }

/* On a "ButtonRelease" event, myButtonNumber stores which
 * mouse button was pressed/released and then we update
 * x/yvelocity accordingly
 * /
    if (myButtRelease) {
        if (myButtonNumber == LEFTMOUSE) {  /* increase
                                               xvelocity */
            if (xvelocity >= 0)
                xvelocity += 3;
            else
                xvelocity -= 3;
        } else if (myButtonNumber == MIDDLEMOUSE) {
                                             /* increase yvelocity*/
            if (yvelocity >= 0)
                yvelocity += 3;
            else
                yvelocity -= 3;
        } else if (myButtonNumber == RIGHTMOUSE) {
                                           /* stop ball */
            xvelocity = yvelocity = 0;
```

```
            } else {
                fprintf(stderr,"ERROR: %s thinks
                        mouse button # ");
                fprintf(stderr,"%d was
                        pressed(?)\n",argv[0],myButtonNumber);
            }
            drawball();
            myButtRelease = FALSE;
        }

        /* On a keypress of Esc key, exit program. */
        if (myKeyPress) {
            if (keysym == XK_Escape)
                clean_exit();
        }

        drawball();

    }
}

static int attributeList[] = { GLX_DOUBLEBUFFER,
                               None };
GLUquadricObj *qobj;

static Bool WaitForNotify(Display *d, XEvent *e, char *arg) {
    return (e->type == MapNotify) && (e->xmap.window ==
                                      (Window)arg);
}

static void openwindow(char *progname) {

    int scrnnum;       /* X screen number  */
    int xorig, yorig;  /* window (upper-left) origin */
    XVisualInfo *vi;
    GLXContext cx;
    Colormap cmap;
    XSizeHints Winhints;/* used to fix window size */
    XSetWindowAttributes swa;
    XColor colorstruct;

    /* Connect to the X server and get screen info */
    if ((dpy = XOpenDisplay(NULL)) == NULL) {
        fprintf(stderr, "%s: cannot connect to X server %s\n",
                            progname, XDisplayName(NULL));
```

```
    exit(1);
}
scrnnum = DefaultScreen(dpy);
ymaxscrn = DisplayHeight(dpy, scrnnum);
xmaxscrn = DisplayWidth(dpy, scrnnum);

/* get an appropriate visual */
vi = glXChooseVisual(dpy, DefaultScreen(dpy),
                     attributeList);
if (vi == NULL) {
    printf("Couldn't get visual.\n");
    exit(0);
}

/* create a GLX context */
cx = glXCreateContext(dpy, vi, None, GL_TRUE);

if (cx == NULL) {
    printf("Couldn't get context.\n");
    exit(0);
}

/* create a colormap */
cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                       vi->visual, AllocAll);

XSync(dpy, 0);
/* create a window */
swa.colormap = cmap;
swa.border_pixel = 0;

/* express interest in certain events */
swa.event_mask = StructureNotifyMask | KeyPressMask |
                 ButtonPressMask |
                 ButtonReleaseMask | ExposureMask;
glwin = XCreateWindow(dpy, RootWindow(dpy, vi->screen),
                      10, 10, 300, 300,
                      0, vi->depth, InputOutput,
                      vi->visual,
                      CWBorderPixel|CWColormap|CWEventMask, &swa);

XMapWindow(dpy, glwin);
XIfEvent(dpy, &event, WaitForNotify, (char*)glwin);

/* connect the context to the window */
```

```
    glXMakeCurrent(dpy, glwin, cx);

   /* express interest in WM killing this app */
    if ((del_atom = XInternAtom(dpy, "WM_DELETE_WINDOW",
                                True)) != None)
        XSetWMProtocols(dpy, glwin, &del_atom, 1);

    colorstruct.pixel = BLACK;
    colorstruct.red   = 0;
    colorstruct.green = 0;
    colorstruct.blue  = 0;
    colorstruct.flags = DoRed | DoGreen | DoBlue;
    XStoreColor(dpy, cmap, &colorstruct);
    colorstruct.pixel = YELLOW;
    colorstruct.red   = 65535;
    colorstruct.green = 65535;
    colorstruct.blue  = 0;
    colorstruct.flags = DoRed | DoGreen | DoBlue;
    XStoreColor(dpy, cmap, &colorstruct);

    glLoadIdentity();
    gluOrtho2D(XMIN - 0.5,  XMAX + 0.5,  YMIN - 0.5,  YMAX + 0.5);

    /* clear the buffer */
    glClearIndex((GLfloat)BLACK);
    qobj = gluNewQuadric();
    gluQuadricDrawStyle(qobj,GLU_FILL);
    glFlush();
}

static void drawball(void) {
    static int xpos = 500, ypos = 500;
    GLdouble radius = 14.0;

    glClear(GL_COLOR_BUFFER_BIT);
    xpos += xvelocity;
    ypos += yvelocity;
    if (xpos > XMAX - radius || xpos < XMIN + radius) {
        xpos -= xvelocity;
        xvelocity = -xvelocity;
    }
    if (ypos > YMAX - radius || ypos < YMIN + radius) {
        ypos -= yvelocity;
        yvelocity = -yvelocity;
    }
```

```
    glIndexi(YELLOW);
    glPushMatrix();
    glTranslatef(xpos,  ypos, 0.);
    gluDisk( qobj, 0., radius, 10, 1);
    glPopMatrix();
    glXSwapBuffers(dpy, glwin);
}

/*  clean_exit  --  Clean up before exiting */
static void clean_exit(void)
{
    gluDeleteQuadric(qobj);
    XCloseDisplay(dpy);
    exit(0);
}
```

Example Two: zrgb.c

The *zrgb.c* example program includes depth buffering. This program won't work on 8-bit IRIS workstations. The IRIS GL version is presented first.

IRIS GL Version of zrgb.c

This is the IRIS GL version of *zrgb.c*. Like *iobounce.c,* this is a pure IRIS GL program.

```
/*                  zrgb.c
 *
 *  This program demostrates zbuffering 3 intersecting RGB
 *  polygons while in doublebuffer mode where,movement of the
 *  mouse with the LEFTMOUSE button depressed will, rotate the 3
 *  polygons. This is done via compound rotations allowing
 *  continuous screen-oriented rotations. (See orient(),
 *  and draw_scene() below).  Notice the effective way there
 *  is no wasted CPU usage when the user moves the mouse out
 *  of the window without holding down LEFTMOUSE--there is no
 *  qtest being performed and so the program simply blocks on
 *  the qread statement. Press the "Esc"[ape] key to exit.
 *  Please note that this program will not work on any 8-bit
 *  IRIS machine.
 *                                          ratman - 1989
```

```
 */

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

Matrix objmat = {
    {1.0, 0.0, 0.0, 0.0},
    {0.0, 1.0, 0.0, 0.0},
    {0.0, 0.0, 1.0, 0.0},
    {0.0, 0.0, 0.0, 1.0},
};

Matrix idmat = {
    {1.0, 0.0, 0.0, 0.0},
    {0.0, 1.0, 0.0, 0.0},
    {0.0, 0.0, 1.0, 0.0},
    {0.0, 0.0, 0.0, 1.0},
};

/* Modes the program can be in */
#define NOTHING 0
#define ORIENT 1

int mode = 0;
int omx, mx, omy, my;  /* old and new mouse position */
float scrnaspect;      /* aspect ratio value         */
long zfar;             /* holds specific machine's   */
                       /* maximum Z depth value      */

main(argc, argv)
int argc;
char *argv[];
{
    long dev;
    short val;
    int redrawneeded=TRUE;  /* Is true when the scene */
                            /* needs redrawing */

    initialize(argv[0]);

    while (TRUE) {

        if (redrawneeded) {
            draw_scene();
```

```
    redrawneeded=FALSE;
}

while (qtest() || (!redrawneeded)) {

    switch(dev=qread(&val)) {

    case ESCKEY:   /* exit when key is going up, */
                   /* not down                   */
        if (val)   /* this avoids the scenario   */
                   /* where a window             */
            break; /* underneath this program's  */
                   /* window--say                */
        exit(0);   /* another copy of this       */
                   /* program--getting the       */
                   /* ESC UP event and exiting   */
                   /* also.                      */
    case REDRAW:
        reshapeviewport();
        redrawneeded=TRUE;
        break;

    case LEFTMOUSE:
        if (val) {
            mode = ORIENT;
            omx = mx;
            omy = my;
        } else
            mode = NOTHING;
        break;

    case MOUSEX:
        omx = mx;
        mx = val;
        if (mode == ORIENT) {
            update_scene();
            redrawneeded=TRUE;
        }
        break;

    case MOUSEY:
        omy = my;
        my = val;
        if (mode == ORIENT) {
            update_scene();
```

```
                    redrawneeded=TRUE;
                }
                break;
            }
        }
    }
}


initialize(progname)
char *progname;
{

    long xscrnsize;   /* size of screen in x used
                       * to set globals  */
    long testifZinst;

    /*
     * This program requires the following to run:
     *  -- z buffer
     *  -- ability to do double-buffered RGB mode
     */
    /* Test for Z buffer */
    testifZinst = getgdesc(GD_BITS_NORM_ZBUFFER);
    if (testifZinst == FALSE) {
         fprintf(stderr,"BUMmer!--%s won't work on ",
                        progname);
         fprintf(stderr,"this machine--zbuffer option not
                        installed.\n");
         exit(0);
    }
    /* Test for double-buffered RGB */
    if (getgdesc(GD_BITS_NORM_DBL_RED) == 0) {
         fprintf(stderr,"BUMmer!--%s won't work on ",
                        progname);
         fprintf(stderr,"this machine--not enough
                        bitplanes.\n");
         exit(0);

    }

    /* Code to keep same aspec ratio as the screen */
    keepaspect(getgdesc(GD_XMMAX), getgdesc(GD_YMMAX));
    scrnaspect =
           (float)getgdesc(GD_XMMAX)/(float)getgdesc(GD_YMMAX);
```

```
    winopen(progname);
    wintitle("Zbuffered RGB #1");

    doublebuffer();
    RGBmode();
    gconfig();
    zbuffer(TRUE);
    glcompat(GLC_ZRANGEMAP, 0);
    zfar = getgdesc(GD_ZMAX);

    qdevice(ESCKEY);
    qdevice(LEFTMOUSE);
    qdevice(MOUSEX);
    qdevice(MOUSEY);
}

update_scene() {

    switch (mode) {

        case ORIENT:
            orient();
            break;
    }
}

orient () {

    pushmatrix();

    loadmatrix(idmat);

    rotate(mx-omx, 'y');
    rotate(omy-my, 'x');

    multmatrix(objmat);
    getmatrix(objmat);
    popmatrix();
}

draw_scene() {

    czclear(0x00C86428, zfar);
```

```
    perspective(400, scrnaspect, 30.0, 60.0);
    translate(0.0, 0.0, -40.0);
    multmatrix(objmat);
    rotate(-580, 'y');   /* skews original view
                          * to show all polygons */
    draw_polys();

    swapbuffers();
}
float polygon1[3][3] = { {-10.0, -10.0,   0.0,},
                         { 10.0, -10.0,   0.0,},
                         {-10.0,  10.0,   0.0,} };

float polygon2[3][3] = { {  0.0, -10.0, -10.0,},
                         {  0.0, -10.0,  10.0,},
                         {  0.0,   5.0, -10.0,} };

float polygon3[4][3] = { {-10.0,   6.0,   4.0,},
                         {-10.0,   3.0,   4.0,},
                         {  4.0,  -9.0, -10.0,},
                         {  4.0,  -6.0, -10.0,} };

draw_polys() {

    bgnpolygon();
    cpack(0x00000000);
    v3f(&polygon1[0][0]);
    cpack(0x007F7F7F);
    v3f(&polygon1[1][0]);
    cpack(0x00FFFFFF);
    v3f(&polygon1[2][0]);
    endpolygon();
    bgnpolygon();
    cpack(0x0000FFFF);
    v3f(&polygon2[0][0]);
    cpack(0x007FFF00);
    v3f(&polygon2[1][0]);
    cpack(0x00FF0000);
    v3f(&polygon2[2][0]);
    endpolygon();

    bgnpolygon();
    cpack(0x0000FFFF);
    v3f(&polygon3[0][0]);
    cpack(0x00FF00FF);
```

```
    v3f(&polygon3[1][0]);
    cpack(0x00FF0000);
    v3f(&polygon3[2][0]);
    cpack(0x00FF00FF);
    v3f(&polygon3[3][0]);
    endpolygon();
}
```

OpenGL Version of zrgb.c

This is the OpenGL version of *zrgb.c*.

```
/*
 *                    zrgb.c
 */
#include <GL/glx.h>
/*
#include <GL/gl.h>
#include <GL/glu.h>
*/
#include <stdio.h>
#include <stdlib.h>
#include <X11/keysym.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>

#define TRUE            1
#define FALSE           0

Display *dpy;         /* The X server connection */
Atom del_atom;        /* WM_DELETE_WINDOW atom   */
Window glwin;         /* handle to the GL window */
XEvent event;

/* function declarations */

static void openwindow(char *);
static void resize_buffer(void);
static void clean_exit(void);
void initGL(void);
void orient(void);
void drawScene(void);
void drawPolys(void);
```

```
static float objmat[16] = {
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0,
};

short ax, ay, az;     /* angles for the "twirling" green
                       * sphere to ride on */
long xsize, ysize;    /* current size-of-window keepers */
long zfar;            /* used in czclear for the machine's
                       * zbuffer max */
long buffermode;      /* flag tracks current window
                       * (single or double) */
double scrnaspect;    /* aspect ratio value */
int xpos, ypos, oxpos, oypos; /* old and new mouse position */


main(argc,argv)
int argc;
char **argv;
{
    int myExpose, myConfigure, myButtPress, myKeyPress;
    int needToDraw = 0;  /* don't set this to true until
                          * we get our first Expose event */

    myExpose = myConfigure = myButtPress = myKeyPress = FALSE;

    openwindow(argv[0]);


   /* start out making the singlebuffer window be
    * our current GL window */
    initGL();     /* do GL init stuff */

    /*
     * The event loop.
     */
    while (1) {      /* standard logic:  get event(s),
                      * process event(s) */

        XEvent event;
        KeySym keysym;
        char buf[4];
```

```
/* this "do while" loop does the 'get events'
 * half of the "get events, process events" action
 * of the infinite while. this is to ensure
 * the event queue is always drained before the events
 * that have come in are processed.
 */

    do {

        XNextEvent(dpy, &event);
            switch (event.type) {

        /* "Expose" events are sort of like "REDRAW" in
         * gl-speak in terms of when a window becomes
         * visible, or a previously
         * invisible part becomes visible.
         */
            case Expose:   /* Exposures */
                needToDraw = myExpose = TRUE;
                break;

        /* "ConfigNotify" events are like "REDRAW" in
         * terms of changes to a window's size or position.
         */
            case ConfigureNotify: /* Resize GL manually */
                xsize = event.xconfigure.width;
                ysize = event.xconfigure.height;
                needToDraw = myConfigure = TRUE;
                break;

        /* Wait for "MotionNotify" events so the
         * queue doesn't fill up
         */
            case MotionNotify:
                myButtPress = TRUE;
                xpos = event.xmotion.x;
                ypos = event.xmotion.y;
                break;

        /* "ClientMessage" is generated if the WM itself
         * is being gunned down and sends an exit signal
         * to any running prog.
         */
            case ClientMessage:
                if (event.xclient.data.l[0] == del_atom)
```

```
                    clean_exit();
                break;

            /* "KeyPress" events are those that would be
             *  generated before whenever queueing up any
                KEYBD key via qdevice.
             */

                case KeyPress:
                   /* save out which unmodified key (i.e. the
                    * key was not modified w/something like
                    * "Shift", "Ctrl", or "Alt") got pressed
                    * for use below.
                    */
                    XLookupString((XKeyEvent *)&event, buf, 4,
                                  &keysym, 0);
                    myKeyPress = TRUE;
                    break;

              }  /* end switch (event.type) */

        } while (XPending(dpy));   /* end "do { } while".
                                    * XPending() is like
                                    * qtest()--it only
                                    * tells you if there're
                                    * any events presently in
                                    * the queue. it does not
                                    * disturb queue's contents
                                    * in any way.
                                    */

    /* On an "Expose" event, redraw the affected pop'd or
     * de-iconized window
     */
        if (myExpose) {
            resize_buffer();
            myExpose = FALSE; /* reset flag--queue now empty */
        }

    /* On a "ConfigureNotify" event, the GL window has either
     * been moved or resized. Respond accordingly and then
     * redraw its contents.
     */
        if (myConfigure) {
            oxpos = xpos;
```

```
            oypos = ypos;
            resize_buffer();
            myConfigure = FALSE;  /* reset flag--queue now
                                   * empty */
        }

        if (needToDraw) {
            drawScene();
            needToDraw = FALSE;
        }

        /* On a keypress of Esc key, exit program.
         */
        if (myKeyPress) {
            if (keysym == XK_Escape)
                clean_exit();
        }

        if (myButtPress) {
            orient();
            drawScene();
            myButtPress = FALSE;
        }
    }      /* end while(1) */

}      /* end main */

static int attributeList[] = { GLX_RGBA,
                               GLX_DOUBLEBUFFER,
                               GLX_RED_SIZE, 1,
                               GLX_GREEN_SIZE, 1,
                               GLX_BLUE_SIZE, 1,
                               GLX_DEPTH_SIZE, 1,
                               None };
static int attributeList2[] = { GLX_RGBA,
                               GLX_RED_SIZE, 1,
                               GLX_GREEN_SIZE, 1,
                               GLX_BLUE_SIZE, 1,
                               GLX_DEPTH_SIZE, 1,
                               None };


static Bool WaitForNotify(Display *d, XEvent *e, char *arg) {
    return (e->type == MapNotify) && (e->xmap.window ==
                                      (Window)arg);
```

```
}

XSizeHints Winhints;          /* used to fix window size */

/* openwindow - establish connection to X server, get screen info, specify the
 * attributes we want the WM to try to provide, and create the GL window */
static void openwindow(char *progname) {

    XVisualInfo *vi;
    GLXContext cx;
    Colormap cmap;
    XSizeHints Winhints;       /* used to fix window size*/
    XSetWindowAttributes swa;
    int scrnnum;               /* X screen number */
    int xorig, yorig;          /* window (upper-left) origin */
    long scrnheight;

   /* define window initial size */
    xorig = 50;  yorig = 40;
    xsize = 300; ysize = 240;
    scrnaspect = xsize / (double) ysize;

   /* Connect to the X server and get screen info */
    if ((dpy = XOpenDisplay(NULL)) == NULL) {
        fprintf(stderr, "%s: cannot connect to X server %s\n",
                                progname, XDisplayName(NULL));
        exit(1);
    }

    scrnnum = DefaultScreen(dpy);
    scrnheight = DisplayHeight(dpy, scrnnum);

        /* get an appropriate visual */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy),
                         attributeList);
    if (vi == NULL) {
        fprintf(stderr, "Unable to obtain visual
                         Doublebuffered visual\n");
        vi = glXChooseVisual(dpy, DefaultScreen(dpy),
                             attributeList2);
    }
    if (vi == NULL) {
        printf("Unable to obtain Singlebuffered
               VISUAL(????)\n");
        exit(0);
```

```
}

/* create a GLX context */
cx = glXCreateContext(dpy, vi, None, GL_TRUE);

/* create a colormap */
cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                       vi->visual, AllocNone);

/* create a window */
swa.colormap = cmap;
swa.border_pixel = 0;
swa.event_mask = StructureNotifyMask  | ButtonPressMask |
                 ExposureMask |
                 Button1MotionMask |
                 KeyPressMask;    /* express interest in
                                   * events */;
glwin = XCreateWindow(dpy, RootWindow(dpy, vi->screen),
                      xorig, yorig, xsize, ysize,
                      0, vi->depth, InputOutput,
                      vi->visual,
                      CWBorderPixel|CWColormap|CWEventMask, &swa);

XMapWindow(dpy, glwin);
XIfEvent(dpy, &event, WaitForNotify, (char*)glwin);

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

if (!(glwin)) {
    fprintf(stderr,"%s: couldn't create 'parent' X
                    window\n",progname);
    exit(1);
}

/* define string that will show up in the window title bar
 * (and icon) */
XStoreName(dpy, glwin, "z-buffered rgb program");

/* specify the values for the Window Size Hints we want to
 * enforce: this window's aspect ratio needs to stay at
 * 1:1, constrain min and max window size, and specify the
 * initial size of the window.
 */
Winhints.width  = xsize;   /* specify desired x/y size of
```

```
                                * window */
    Winhints.height = ysize;
    Winhints.min_width = xorig;  /* define min and max */
    Winhints.max_width = scrnheight-1;  /* width and height */
    Winhints.min_height = yorig;
    Winhints.max_height = scrnheight-1;
    Winhints.min_aspect.x = xsize;    /* keep aspect to a xsize:ysize ratio */
    Winhints.max_aspect.x = xsize;
    Winhints.min_aspect.y = ysize;
    Winhints.max_aspect.y = ysize;
    /* set the corresponding flags */
    Winhints.flags = USSize|PMaxSize|PMinSize|PAspect;
    XSetNormalHints(dpy, glwin, &Winhints);

    /* express interest in WM killing this app */
    if ((del_atom = XInternAtom(dpy, "WM_DELETE_WINDOW",
                                True)) != None)
        XSetWMProtocols(dpy, glwin, &del_atom, 1);

    return ;
}

/*  window has been moved or resized so update viewport & CTM stuff. */
static void resize_buffer() {

    XSync(dpy, False);  /* STILL NEED THIS????? */
                        /* Need before GL reshape */
    scrnaspect = xsize / (double) ysize;
    glViewport(0, 0, (short) (xsize-1), (short) (ysize-1));
}

/*  clean up before exiting */
static void clean_exit(void)
{
    XCloseDisplay(dpy);
    exit(0);
}

/* setup all necessary GL initialzation parameters. */
void initGL()
{
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.16, 0.39, 0.78, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
```

```
    gluPerspective(400.0, scrnaspect, 30.0, 1000.0);
}

void orient()
{
    float dx, dy;
    glPushMatrix();
    dx = xpos-oxpos;
    dy = oypos-ypos;
    glLoadIdentity();
    glRotatef((float) (0.03*(xpos-oxpos)), 1.0, 0.0, 0.0);
    glRotatef((float) (0.03*(oypos-ypos)), 0.0, 1.0, 0.0);
    glMultMatrixf(objmat);
    glGetFloatv(GL_MODELVIEW_MATRIX, objmat);

    glPopMatrix();
}

void drawScene()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
    glTranslatef(0.0, 0.0, -40.0);
    glMultMatrixf(objmat);
    glRotatef(-220.0, 0.0, 1.0, 0.0);  /* skews orig view to
                                        * show all polys */
    drawPolys();
    glPopMatrix();
    glFlush ();
    glXSwapBuffers(dpy, glwin);
}

float polygon1[3][3] = { {-10.0, -10.0,   0.0,},
                         { 10.0, -10.0,   0.0,},
                         {-10.0,  10.0,   0.0,} };

float polygon2[3][3] = { {  0.0, -10.0, -10.0,},
                         {  0.0, -10.0,  10.0,},
                         {  0.0,   5.0, -10.0,} };

float polygon3[4][3] = { {-10.0,   6.0,   4.0,},
                         {-10.0,   3.0,   4.0,},
                         {  4.0,  -9.0, -10.0,},
                         {  4.0,  -6.0, -10.0,} };
```

```
void drawPolys()
{
    glBegin(GL_POLYGON);
    glColor4f(0.0, 0.0, 0.0, 0.0);
    glVertex3fv(&polygon1[0][0]);
    glColor4f(0.5, 0.5, 0.5, 0.0);
    glVertex3fv(&polygon1[1][0]);
    glColor4f(1.0, 1.0, 1.0, 0.0);
    glVertex3fv(&polygon1[2][0]);
    glEnd();

    glBegin(GL_POLYGON);
    glColor4f(1.0, 1.0, 0.0, 0.0);
    glVertex3fv(&polygon2[0][0]);
    glColor4f(0.0, 1.0, 0.5, 0.0);
    glVertex3fv(&polygon2[1][0]);
    glColor4f(0.0, 0.0, 1.0, 0.0);
    glVertex3fv(&polygon2[2][0]);
    glEnd();

    glBegin(GL_POLYGON);
    glColor4f(1.0, 1.0, 0.0, 0.0);
    glVertex3fv(&polygon3[0][0]);
    glColor4f(1.0, 0.0, 1.0, 0.0);
    glVertex3fv(&polygon3[1][0]);
    glColor4f(0.0, 0.0, 1.0, 0.0);
    glVertex3fv(&polygon3[2][0]);
    glColor4f(1.0, 0.0, 1.0, 0.0);
    glVertex3fv(&polygon3[3][0]);
    glEnd();

}
```

Index

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1797-030.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389