The OpenGL Porting Guide

Document Number 007-1797-020

CONTRIBUTORS

Written by C J Silverio, Beth Fryer, and Jed Hartman
Edited by Nancy Schweiger
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz
Production by Lorrie Williams
Engineering contributions by Kurt Akeley, Allen Akin, Gavin Bell, Derrick Burns, Dave Ciemiewicz, Tom Davis, Chris Frazier, Phil Karlton, Reuel Nash, Mark Segal, Dave Shreiner, Rolf van Widenfelt, and Mason Woo

The OpenGL Porting Guide
Document Number 007-1797-020

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 automatic translation script (called *toogl*), lists OpenGL equivalents for IRIS GL calls, describes how to reimplement IRIS GL windowing code with X and IRIS IM™ (IRIS IM is Silicon Graphics' port of the industry-standard OSF/Motif™ software), and gives you the basics of what you need to know about X.

This book is intended 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, refer to the *OpenGL Programming Guide*, available from Addison-Wesley.

Note: This book is written for programmers who are working in C. The Fortran and Ada wrappers for OpenGL have not yet been determined, but will be available sometime in the future.

How to Use This Guide

As you use this guide, you'll probably need to refer often to the OpenGL reference pages, as well as the IRIS GL reference pages and programming guide. 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 or from Silicon Graphics.

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 tools Silicon Graphics® provides to help you with the translation, and provides some general porting instructions.
- Chapter 2, “Using toogl,” describes 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, “Using the Auxiliary Library to Replace Windowing and Event Handling Calls,” explains how to use the auxiliary library to replace simple windowing and event handling commands (rather than using Xt or Xlib).
- Chapter 5, “Mixed-Model Programming,” describes two methods for using the X Window System™ to manage windows and events with OpenGL: using Xt and the Silicon Graphics mixed-model programming widget or using Xlib. IRIS IM is also discussed in this chapter.
- 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 is offered in Chapter 1.
- Appendix C, “Some Important OpenGL Basics,” explains the OpenGL naming conventions, lists OpenGL defined types, and describes error handling in OpenGL.
- Appendix D, “Example OpenGL Program with the Auxiliary Library,” provides an example OpenGL program that uses the auxiliary library for windowing and event handling.

- Appendix E, “Example Mixed-Model Program with WorkProc,” provides an example OpenGL mixed-model program using Xt, IRIS IM, and the Silicon Graphics widget. The program demonstrates the use of 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

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

- *OpenGL Reference Manual*, from the OpenGL Architecture Review Board, published by Addison-Wesley Publishing Company, Reading, Massachusetts, 1992. ISBN 0-201-63276-4.
- *OpenGL Programming Guide*, written by Jackie Neider, Tom Davis, and Mason Woo, published by Addison-Wesley Publishing Company, Reading, Massachusetts. ISBN 0-201-63274-8

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

- *Graphics Library Programming Guide*
- *Graphics Library Programming Tools and Techniques*

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

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

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

- Volume One: *Xlib Programming Manual*, by Adrian Nye, published by O'Reilly & Associates, Inc., Sebastopol, California.
- Volume Four: *X Toolkit Intrinsics Programming Manual*, by Adrian Nye and Tim O'Reilly, published by O'Reilly & Associates, Inc., Sebastopol, California.
- Volume Five: *X Toolkit Intrinsics Reference Manual*, published by O'Reilly & Associates, Inc., Sebastopol, California.

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.

Style Conventions

These style conventions are used in this guide:

Bold—Function names.

Italics—IRIX command names, variables, arguments, parameter names, and spatial dimensions.

Code examples are set off from the text in a fixed-width typeface. All code examples are in C.

A family of functions whose names all start out the same way are referred to collectively using the common name-beginning and an asterisk. For instance, **glColor*()** refers collectively to **glColor3b()**, **glColor4usv()**, and all the other functions whose names begin with the string "glColor."

Chapter 1

Introduction to Porting from IRIS GL to OpenGL

This chapter provides an overview to porting from IRIS GL to OpenGL. It lists the most important differences between OpenGL and IRIS GL, describes some tools and libraries provided by Silicon Graphics that might help you port your code, and lists the basic steps for porting code from IRIS GL to OpenGL. It also provides some advice for programmers who do not plan to port to OpenGL immediately.

What's Different?

Due in part to OpenGL's focus on portability, OpenGL and IRIS GL differ in several major areas. This section lists a few very important ways in which OpenGL is different from IRIS GL. A more complete list of the differences between the two is provided in Appendix B, "Differences Between OpenGL and IRIS GL."

Here are some key differences between OpenGL and IRIS GL:

- Since OpenGL is window-system independent, it contains no windowing, pop-up menus, event handling, color-map loading, buffer allocation / management, font file formats, or cursor handling. These functions are delegated to the window or operating system. Silicon Graphics provides a small, auxiliary library that you can use to replace simple windowing, event handling, and color-map loading calls. See Chapter 4, "Using the Auxiliary Library to Replace Windowing and Event Handling Calls," for more information. If you need more sophisticated windowing and event handling calls, you'll probably need to turn your program into a mixed-model program (a program that mixes X and GL calls). Silicon Graphics provides some special OpenGL / X calls and a OpenGL / X widget to help you replace your IRIS GL windowing, event, and color-map handling calls. See Chapter 5, "Mixed-Model Programming," for details.

- OpenGL establishes and adheres to a standard “name space.” OpenGL commands begin with the gl prefix (**glEnable()**, **glTranslatef()**, and so on). This prevents conflict with commands from other libraries. “OpenGL Command Names” on page 179 explains the OpenGL naming conventions, and “OpenGL Defined Types” on page 181 lists the OpenGL defined types with their C data type equivalents.
- Like IRIS GL, OpenGL maintains many state variables (for color, fog, texture, lighting, viewport, and so on). But OpenGL manages state variables more directly and consistently than does IRIS GL. With OpenGL there are no tables—you just load values directly. Since OpenGL doesn’t keep tables of lights and materials that you’ve predefined, 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 explains how to do this. You might also refer to “Porting Lighting and Materials Calls” on page 68 and “Porting Texture Calls” on page 73 for more discussion and some examples.
- OpenGL display lists are not editable. In OpenGL, the sole purpose of display lists is to efficiently cache OpenGL commands. This means that IRIS GL calls for editing display lists have no OpenGL equivalents—if your IRIS GL code edits display lists, you’ll need to reimplement to some extent. “Porting Display Lists” on page 63 lists the relevant IRIS GL calls, and “Edited Display Lists” on page 65 provides some suggestions for porting code that edits display lists.
- 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 112 provides suggestions for porting fonts and strings.
- 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 comprising the GLU 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 some tools and libraries to help you port your code:

- The *toogl* tool translates your program's IRIS GL calls to OpenGL calls. *toogl* can't translate everything (in particular, it can't translate windowing and event calls), so you have to edit the output—but it does do a lot of the translation work for you. Chapter 2, "Using toogl," explains how to use this tool.
- 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 5 explains how to do this. Reference pages for the GLX routines are included in the *OpenGL Reference Manual*. You might want to look at the glXIntro reference page first.
- The GLwDrawingArea and GLwMDrawingArea widgets help you port your code to mixed-model mode. (In the context of this book, a mixed-model program is a mixture of X and OpenGL, in which OpenGL is used for rendering and X is used for windowing, event handling, fonts, and so on.) 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 5.
- The auxiliary library was designed to support the code examples in the *OpenGL Programming Guide*. It is intentionally very simple, but it does provide some simple routines that you can use to open windows, detect input, load the color map, render 3-D objects, and so on. For information on how to use the auxiliary library, see the *OpenGL Programming Guide* and Chapter 4 of this book.

How to Port Your Code to OpenGL

This section lists three cases—select the one that best matches your situation and complete the porting tasks listed (you don't need to do them in any particular order).

Case 1: Your IRIS GL code is already in mixed-model mode. (A mixed-model program uses GL for rendering and X for all window system calls, including windowing and event handling.)

This means it'll be much easier to port to OpenGL. Here's what you'll need to do:

- Run your program through a C beautifier (such as *cb*), then run the *toogl* filter script on your code. Spend some time editing its output. See Chapter 2 for a list of known trouble spots where hand-porting is required. You'll probably need to hand-port some of the trickier commands—see Chapter 3, "After toogl: How to Finish Porting to OpenGL," for specific suggestions.
- Convert your IRIS GL/X calls to OpenGL/X calls. If you used one of the mixed-model widgets, GlxDraw or GlxMDraw, switch to the OpenGL version: GLwDrawingArea or GLwMDrawingArea. 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. Chapter 5 discusses mixed-model programming in OpenGL and provides information about the OpenGL version of the Silicon Graphics mixed-model widget.

Case 2: Your IRIS GL code is not in mixed-model mode, but your windowing, color map, and event handling code is reasonably simple, conforms to Silicon Graphics recommendations, and does not use unsupported calls.

You can use the *OpenGL Programming Guide* auxiliary library to replace your IRIS GL windowing, color map, and event handling calls. Here's what you'll need to do:

- Replace windowing and event handling calls with auxiliary library calls. See Chapter 4 for porting instructions and refer to the *OpenGL Programming Guide* for more information on the auxiliary library.
- Run your program through a C beautifier (such as *cb*), then run the *toogl* filter script on your code. Spend some time editing its output. See Chapter 2 for a list of known trouble spots where hand-porting is required. You'll probably need to hand-port some of the trickier commands—see Chapter 3 for specific suggestions.

Case 3: Your IRIS GL code is not in mixed-model mode and your windowing and event handling code uses unsupported calls, does not conform to Silicon Graphics recommendations, or is especially complicated or unusual in scope.

In this case, you're probably better off porting to mixed-model mode. Here's what you'll need to do:

- Run your program through a C beautifier (such as *cb*), then run the *toogl* filter script on your code. Spend some time editing its output. See Chapter 2 for a list of known trouble spots where hand-porting is required. You'll probably need to hand-port some of the trickier commands—see Chapter 3 for specific suggestions.
- Port your program to the mixed-model mode (mixing 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 5 for more information.

In all cases, after you've finished the porting tasks listed, you'll probably need 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 might find it useful to refer to "Error Handling" on page 181 which gives some basic information on error handling in OpenGL.

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 very 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 the *Graphics Library Programming Tools and Techniques* manual for detailed instructions.

Another thing you can do now is to 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.)

Finally, replace any obsolete or unsupported calls with newer IRIS GL equivalents as soon as possible.

Note: You might also consider switching now from IRIS GL to Open Inventor™, a powerful toolkit based on OpenGL. You can order Open Inventor from your Silicon Graphics sales representative or through Silicon Graphics Software Express.

Chapter 2

Using *toogl*

This chapter documents 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.

How to Get and Call *toogl*

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. 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 will need to port by hand.

You can get a copy of *toogl* from */usr/people/4Dgifts/bin*. If you want to look at the source (it's in C++), you can get a copy from the OpenGL directory in */usr/people/4Dgifts*. (If you have any problems finding something in 4Dgifts, refer to the */usr/people/4Dgifts/README* file, which explains the structure and contents of 4Dgifts.)

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, replacing them with warning comments. The **-w** and **-q** flags merely suppress the comments.

Keep your original source! Accidents happen.

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

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

Using *gdiff* to Compare Files

You'll probably find the *gdiff* command very helpful when editing *toogl* output. *gdiff* allows you to easily see differences between the *toogl* output and your original program—or any other version of the program. To use *gdiff*, type:

```
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.

See the *gdiff* reference page for more information.

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 it finds an IRIS GL function, it replaces the function with the corresponding OpenGL function(s).

Since *toogl* can't translate everything, you need to edit its output. Any time *toogl* translates code that you might need 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.)

Hints for 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* on it.
- Use *gdiff* to browse through your 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, this code:

  ```
  v3f
  (foo);
  ```

  will be left unchanged, as will:

  v3f /* comment */ (foo);

  (Running a C beautifier on your program before using *toogl* can prevent this sort of problem.)
- C comments inside the argument list of a function shouldn’t contain parentheses or quote characters. Faced with the following code, *toogl* will generate a warning and do no translation:

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

Editing *toogl* Output: Areas that Need Special Attention

After you’ve run *toogl* on your code, you need 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 at all. 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.

Windowing, Device, and Event Calls

toogl can't translate sections of code where you make window manager, window configuration, device, or event calls, or where you load a color map. You'll need to rewrite these yourself. You can use the **-w** and **-q** options to make *toogl* leave this code alone, so you can still read it to translate it manually. If your windowing and event handling calls are very simple and straightforward, you can replace them with calls from the auxiliary library. Chapter 4, "Using the Auxiliary Library to Replace Windowing and Event Handling Calls," explains how to do this. If your windowing and event handling calls are fairly sophisticated, you'll need to switch to mixed-model programming—this is explained in Chapter 5, "Mixed-Model Programming."

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'll need to port them by hand.

clear() Calls

Check to make sure **clear()** calls are correctly translated. For example, you might have code that clears the window this way:

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

toogl translates those two lines into:

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

This code is incorrect, unfortunately, because OpenGL maintains a clear color, distinct from its drawing color—also, OpenGL does not provide the color constant, BLACK. A better translation would be:

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

Get Calls

toogl doesn't 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 get 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 18.

rotate()

The OpenGL rotation call, **glRotate()**, is somewhat different from **rotate()**. You'll probably need to massage the code a bit after translating with *toogl*. See "Porting Matrix and Transformation Calls" on page 22 for details.

swaptmesh()

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

Texturing

toogl correctly translates texture coordinate calls, but that's about it. You'll need to do some additional work by hand. "Porting Texture Calls" on page 73 explains how.

defs/binds

OpenGL doesn't keep tables of lights and materials that you've predefined, so it has no equivalent for "binds." You can mimic this behavior by using display lists. See "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67 for more information. You might also look at "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

toogl can't directly translate some IRIS GL calls into OpenGL calls. **arcf()** is one example of such a call. You'll need to port such calls by hand. "Editing toogl Output: An Example" on page 14 gives an example of how you might hand-port 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 unsupported in OpenGL.

Performance

toogl doesn’t necessarily produce fast OpenGL code; in fact, there are several aspects to an automatic port of this kind which are known to result 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” and “Porting Lighting and Materials Calls” in Chapter 3.

Two features of OpenGL which 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. Note that if you aren’t careful, it’s possible to set up indirect rendering without noticing that it’s indirect; specify direct rendering explicitly where possible.

A few more tips:

- If you’re drawing independent triangles, there’s no need to put **glBegin()** and **glEnd()** around each set of three vertices; simply call **glBegin(GL_TRIANGLES)** and then list as many separate triangles as you need before the **glEnd()**. This optimization alone can improve performance tremendously.
- If you aren’t using the z-buffer, be sure to disable it. This is particularly important when you call **glDrawPixels()** or other non-3D drawing functions.
- Be sure to disable texturing when calling **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

Here's an example of how you might handle a call like **arcf()**. *toogl* translates this 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 need to fill in its arguments by hand. Compare the reference pages for the two commands, or refer to the section in this guide that discusses porting that command (in this case, "Arcs and Circles" on page 41). Perusal of that material will tell you 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, and the second angle is 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 manage to translate fully. To get the most out of this discussion, refer to the reference pages as necessary. (If you want a printed version of the reference pages, buy the *OpenGL Reference Manual*, described in the introductory section "About This Guide" at the beginning of this guide.)

Header Files

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

Your IRIS GL program probably uses these include lines:

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

In your OpenGL program, you'll need to replace these with:

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

#include <Xm/Xm.h>          /* (These are X header files-- */
#include <Xm/Frame.h>       /* you don't need them if */
#include <Xm/Form.h>        /* you're using the auxiliary */
#include <X11/StringDefs.h> /* library instead of */
#include <X11/keysym.h>     /* mixed-model.) */
```

If you use the auxiliary library, you'll also need to include:

```
#include "aux.h"
```

If your IRIS GL program uses the mixed-model widget, it uses one of these include lines:

- For the IRIS IM version of the widget:

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

- For the generic version of the widget:

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

For the OpenGL version, substitute these include lines:

- For the IRIS IM version of the widget:

  ```
  #include <GL/GLwMDrawA.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. The command:

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

takes a bitwise OR of symbolic constants, indicating which groups of state variables to push onto an attribute stack. Each constant refers to a group of state variables. Table 3-1 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 reference page for **glPushAttrib()**.

Table 3-1 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
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

To restore the values of the state variables to those saved with the last **glPushAttrib()**, simply call **glPopAttrib()**. The variables you didn't save will remain unchanged. The attribute stack has a finite depth of at least 16.

Porting IRIS GL 'Get' Commands

"Get" calls in IRIS GL were of the form:

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

Your IRIS GL code probably includes calls that look something like:

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

Gets in OpenGL use **glGet*()** commands and look something like this:

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

Table A-1 lists the IRIS GL get commands with their OpenGL equivalents.

In general, this guide lists various parameters for **glGet*()** functions 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 22.

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

About glGet*()

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

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

- **glGetDoublev()**

The commands have the syntax:

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

where *value* is of type *GLenum* and *data* of type *GLdatatype*. If you issue a **glGet*()** command that returns types different from the type expected, the type is converted appropriately. For a complete list of **glGet*()** parameters, see the reference page.

glGet() Conventions Used in This Book

This guide, for the sake of brevity, usually shortens the reference to the form **glGet(GL_GET_TYPE)**. For example,

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

will be abbreviated as:

```
glGet(GL_VIEWPORT);
```

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

Porting Commands that Required Current Graphics Position

OpenGL does not 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 has been discouraged. You will need to reimplement 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 IRIS GL's 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()**.

Porting notes:

- 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 doing a clear.
- Since *toogl* has no concept of context, it will not correctly translate color calls immediately preceding clears into **glClearColor()** calls. You will have to do this by hand. For example, you might have cleared your viewport with code like this:

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

  toogl will translate those two lines into:

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

  That fragment might correctly read:

  ```
  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 3-2 lists the various clear calls with their IRIS GL equivalents.

Table 3-2 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 combine them into a single **glClear()** call—this might improve your program's performance.

Porting Matrix and Transformation Calls

Porting notes:

- There is no single-matrix mode. You are always in double-matrix mode in OpenGL.
- Angles are now measured in degrees, instead of tenths of degrees.
- Projection matrix calls, like **glFrustum()** and **glOrtho()**, now multiply onto the current matrix, instead of being loaded onto the current matrix.
- The OpenGL call **glRotate()** is very different from **rotate()**. You can now rotate around any arbitrary axis, instead of being confined to the *x*, *y*, and *z* axes. But you will probably have to port **rotate()** calls by hand, since *toogl* often has trouble translating them. 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 would be:

  ```
  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.

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

Figure 3-1 A 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 A Generic OpenGL Translation

The important thing is that this is a change in *documentation only*—OpenGL matrices are completely compatible to 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 like this:

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

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

Figure 3-3 An OpenGL Matrix Example

- OpenGL has no equivalent to the **polarview()** call. You can replace such a call easily, however, with a translation and three rotations:

  ```
  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* does not translate this call correctly, so you will have to port by hand. See the reference page for **gluLookAt()** for more information.

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

Table 3-3 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
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 2-dimensional orthographic projection matrix
perspective()	gluPerspective()	define a perspective projection matrix
picksize()	gluPickMatrix()	define a picking region

Table 3-3 (continued) Matrix Operations

IRIS GL Call	OpenGL Call	Meaning
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 point specified, 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 3-4 lists the mode types available for arguments to **glMatrixMode()**. The corresponding **mmode()** arguments are listed in the second column.

Table 3-4 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

Note that 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 code might have used it. If it did, you will need to reimplement. OpenGL is always in double-matrix mode, and is initially in GL_MODELVIEW mode.

Most IRIS GL code in MSINGLE mode looks like this:

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

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

```
...
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 3-5 lists related gets.

Table 3-5 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

Table 3-5 (continued) Arguments for Transformation Matrix Queries

IRIS GL Matrix Query	OpenGL glGet() Matrix Query	Meaning
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 the modelview matrix stack
—	GL_MAX_PROJECTION_STACK_DEPTH	return maximum supported depth of the projection matrix stack
—	GL_MAX_TEXTURE_STACK_DEPTH	return maximum supported depth of the texture matrix stack
—	GL_MODELVIEW_STACK_DEPTH	returns number of matrices on the modelview stack

Table 3-5 (continued) Arguments for Transformation Matrix Queries

IRIS GL Matrix Query	OpenGL glGet() Matrix Query	Meaning
—	GL_PROJECTION_STACK_DEPTH	returns number of matrices on the projection stack
—	GL_TEXTURE_STACK_DEPTH	returns number of matrices on the texture stack

Viewports, Screenmasks, and Scrboxes

The following IRIS GL calls have no OpenGL equivalent:

- **reshapeviewport()** (see "Handling Redraw Events" on page 99 for information on how to replace this with auxiliary library calls)
- **scrbox()**, **getscrbox()**

Porting notes: With **viewport()**, you specified the *x* coordinates (in pixels) for the left and right of the viewport rectangle and the *y* coordinates for the top and bottom. With **glViewport()**, however, you specify the *x* and *y* coordinates (in pixels) of the lower left corner of the viewport rectangle along with its width and height.

Table 3-6 lists the OpenGL equivalents for viewport commands.

Table 3-6 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)	returns viewport dimensions

Clipping Planes

OpenGL implements clipping planes the way IRIS GL did, though you can now also query clipping planes. Table 3-7 lists the OpenGL equivalents to IRIS GL calls.

Table 3-7 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()	returns clipping plane equation
—	glIsEnabled(GL_CLIP_PLANE*i*)	returns true if clip plane *i* is enabled
scrmask()	glScissor()	defines 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.

The IRIS GL Sphere Library

The sphere library that worked with IRIS GL isn't yet available for OpenGL. Look for it in a later release of OpenGL. In the meantime, you can replace

your sphere library calls with quadrics routines from the GLU library. Refer to the *OpenGL Programming Guide* and the GLU reference pages in the *OpenGL Reference Manual* for details on using the GLU library. Table 3-8 summarizes OpenGL quadrics calls.

Table 3-8 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 following code fragment uses two quadrics objects to draw four quadrics, two of them textured.

```
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);
```

The v() Commands

In IRIS GL, you use variations on the **v()** call to specify vertices. This call has a direct successor in OpenGL, **glVertex()**:

```
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.

bgn/end Commands

IRIS GL uses the begin/end paradigm but has a different call for each graphics primitive. For example, you probably used **bgnpolygon()** and **endpolygon()** to draw polygons, and **bgnline()** and **endline()** to draw lines. With the OpenGL, you use the **glBegin()**/**glEnd()** structure for both. (The 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 3-9 below, along with the IRIS GL calls they replace (if any).

Table 3-9 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

Table 3-9 (continued) Calls for Drawing Primitives

IRIS GL Call	Value of glBegin() Mode	Meaning
—	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 “Triangles” on page 40.

There is no limit to the number of vertices you can specify between a **glBegin()** / **glEnd()** pair.

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 3-10 lists the commands valid inside a **glBegin()** / **glEnd()** pair.

Table 3-10 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

Table 3-10 (continued) Valid Commands inside a Begin/End Structure

IRIS GL Call	OpenGL Equivalent	Meaning
—	glEdgeFlag()	control drawing edges
lmbind()	glMaterial()	set material properties

If you use any other OpenGL command inside a **glBegin()**/**glEnd()** pair, you'll get unpredictable results, or possibly an error.

Points

OpenGL has no command to draw a single point. Otherwise, porting point calls is straightforward. Table 3-11 lists commands for drawing points.

Table 3-11 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
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 get commands.

Lines

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

Table 3-12 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)

Note that 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. You might have used these calls: **draw()**, **lsbackup()**, **getlsbackup()**, **resetls()**, **getresetls()**. If so, you will have to reimplement.

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

Polygons and Quadrilaterals

Porting notes:

- There is no direct equivalent for **concave(TRUE)**. You might want to use the tessellation routines in the GLU, described in "Tessellated Polygons" on page 40.
- 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 them, you'll have to reimplement using **glBegin(GL_POLYGON)**. Table 3-9 lists the OpenGL equivalents to IRIS GL polygon drawing calls.

Table 3-13 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
—	glEdgeFlag()	
polymode()	glPolygonMode()	set polygon drawing mode

Table 3-13 (continued) Calls for Drawing Polygons

IRIS GL Call	OpenGL Equivalent	Meaning
rect(), rectf(),	glRect()	draw a rectangle
sbox(), sboxf()	—	draw a screen-aligned rectangle

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 (the back or the front) that the mode applies to. Its syntax is:

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

where *face* is one of:

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 3-14 lists IRIS GL polygon modes and the corresponding OpenGL modes.

Table 3-14 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

Polygon Stipples

Porting notes:

- 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 32x32 bit pattern.
- Stipple encoding is affected by **glPixelStore()**. See “Porting Pixel Operations” on page 46 for more information.

Table 3-15 lists polygon stipple calls.

Table 3-15 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()**.

Here’s an example OpenGL code fragment that demonstrates polygon stippling:

```
/*  polys.c  */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.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 ();
}
```

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 3-16 lists the calls for drawing tessellated polygons.

Table 3-16 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 in a contours
gluNextContour()	indicate that the next series of vertices describe a new contour
gluEndPolygon()	end the polygon specification

For complete details, see the reference pages for the commands in Table 3-16.

Triangles

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

Porting notes:

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

Table 3-17 lists the commands for drawing triangles.

Table 3-17 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

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 3-18 lists the IRIS GL arc and circle commands with the corresponding OpenGL (GLU) commands.

Table 3-18 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 very different from the **arc()** call. Refer to the **gluPartialDisk()** reference page for complete information.

You can do some things with OpenGL arcs and circles that you can't do with IRIS GL. Refer to the *OpenGL Programming Guide* and the reference pages in the *OpenGL Reference Manual* for detailed information on OpenGL arcs and circles (which are called disks and partial disks in OpenGL).

Porting notes:

- 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*.
- The sweep angle is now clockwise instead of counterclockwise, as shown in Figure 3-4.

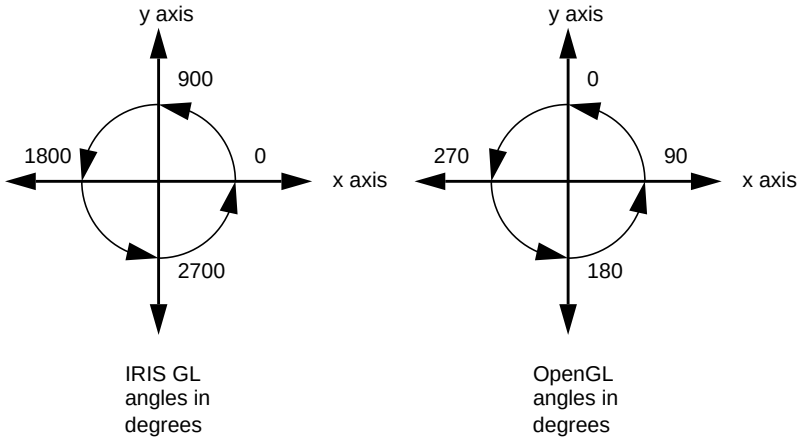

Figure 3-4 Drawing Angles: IRIS GL vs. OpenGL

Spheres

Porting notes:

- You can no longer control the type of primitives used to draw the sphere. You have control of drawing precision in another way: you can use 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 "The IRIS GL Sphere Library" on page 29 for more information about replacing sphere library calls.

Table 3-19 lists the IRIS GL calls for drawing spheres along with the corresponding GLU calls where available.

Table 3-19 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

The major change you'll have to watch out for is the change in how color maps are implemented.

Porting notes:

- Though you can set color map indices with the OpenGL **glIndex()** call, OpenGL does not provide a routine for loading color map indices. If you're using the auxiliary library, see "Using Color-Index Mode" on page 106 for instructions on loading color maps. If you're using X, see "X Color Maps" on page 125 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'll need to port by hand.
- Some calls to **c()** or **color()** might need to be translated to **glClearColor()** or **glClearIndex()** and not **glColor()** or **glIndex()**. See "Porting Screen and Buffer Clearing Commands" on page 20 for details.
- The RGBA writemask is not for each bit, just each component.
- IRIS GL provided 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'll need to port them by hand.

Color Calls

Table 3-20 lists equivalent color calls.

Table 3-20 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()	auxSetOneColor() or XStoreColor()	see "Using Color-Index Mode" on page 106 or "X Color Maps" on page 125
RGBcolor()	glColor()	sets RGB color
writemask()	glIndexMask()	sets the color index mode color mask

Table 3-20 (continued) Color Calls

IRIS GL Call	OpenGL Call	Meaning
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, you'll need to implement them using X's colormap facilities. Therefore, **multimap()**, **onemap()**, **getcmmode()**, **setmap()**, and **getmap()** have no OpenGL equivalents.

Shading Models

As with IRIS GL, you can switch between smooth (Gouraud) shading and flat shading. Table 3-21 lists the calls.

Table 3-21 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 on dithering on / off

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

Porting Pixel Operations

Porting notes:

- 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. Note that 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 gives you some additional flexibility in pixel operations. Table 3-22 lists calls for pixel operations.

Table 3-22 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

Table 3-22 (continued) Pixel Operations

IRIS GL Call	OpenGL Call	Meaning
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:

```
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

Porting notes:

- The fog calls have been restructured, so you might have to rewrite them by hand. 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, use **glHint(GL_FOG_HINT)**.

Table 3-23 lists the IRIS GL calls for managing fog along with the corresponding OpenGL calls.

Table 3-23 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 3-24 lists the arguments you can pass to **glFog()**.

Table 3-24 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 3-25

The fog density argument of OpenGL is different than the fog density argument of IRIS GL. 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, and *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 3-25 lists the OpenGL equivalents for IRIS GL fog modes.

Table 3-25 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)

Here's an example program that demonstrates 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 <GL/gl.h>
#include <GL/glu.h>
#include "aux.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_LEQUAL);
    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);
    auxWireIcosahedron(1.0);
    glFlush();
}

void myReshape(GLsizei w, GLsizei 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 obj. into view*/
}
/*  Main Loop */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA |
                        AUX_DEPTH);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Curve and Surface Commands

OpenGL does not support equivalents to the old-style curves and surface patches. You'll need to reimplement if your code 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 the *OpenGL Programming Guide* 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 3-26 lists the NURBS object commands.

Table 3-26 Calls for Managing NURBS Objects

OpenGL Call	Meaning
gluNewNurbsRenderer()	create a new NURBS object
gluDeleteNurbsRenderer()	delete a NURBS object
gluNurbsCallback()	associate a callback with a NURBS object, for error handling

Porting notes:

- 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 very 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 3-27 summarizes the calls for drawing NURBS curves.

Table 3-27 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 3-28 lists NURBS curve types.

Table 3-28 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 reference page for **glMap1()**.

Trimming Curves

OpenGL trimming curves are very similar to IRIS GL trimming curves. Table 3-29 lists the calls for defining trimming curves.

Table 3-29 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 3-30 summarizes the calls for drawing NURBS surfaces.

Table 3-30 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 3-31 lists parameters for surface types.

Table 3-31 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 3-31 (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 reference page for **glMap2()**.

Here's an example program that draws a trimmed 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 <GL/gl.h>
#include <GL/glu.h>
#include "aux.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;
        }
    }
}

/*  Initialize material property and depth buffer.
 */
void myinit(void)
{
    GLfloat mat_diffuse[] = { 0.6, 0.6, 0.6, 1.0 };
    GLfloat mat_specular[] = { 0.9, 0.9, 0.9, 1.0 };
    GLfloat mat_shininess[] = { 128.0 };

    glClearColor (0.0, 0.0, 0.0, 1.0);
    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);
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);

    init_surface();

    theNurb = gluNewNurbsRenderer();
    gluNurbsProperty(theNurb, GLU_SAMPLING_TOLERANCE,
                     50.0);
    gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL);
}

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] = /* counterclockwise */
    {{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 myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, 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);
}

/*  Main Loop */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA |
                        AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Antialiasing Calls

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'll need to allocate destination alpha bits when you choose a visual—you need to use X for this, so refer to Chapter 5.

Porting 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 3-32 lists IRIS GL and OpenGL blending calls.

Table 3-32 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 3-33 lists the OpenGL equivalents to the IRIS GL blend factors.

Table 3-33 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
BF_MDC	GL_ONE_MINUS_DST_COLOR	source only
BF_MIN_SA_MDA	GL_SRC_ALPHA_SATURATE	

afunction() Test Functions

Table 3-34 lists the available alpha test functions.

Table 3-34 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 IRIS GL's antialiasing calls. Table 3-35 lists them.

Table 3-35 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.

With 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);
```

where *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.

IRIS GL also allowed end correction by calling:

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

OpenGL doesn't have an equivalent for this call.

Accumulation Buffer Calls

You must allocate your accumulation buffer by requesting the appropriate visual with **auxInitDisplayMode()** or **glXChooseVisual()**. (To learn how to use **auxInitDisplayMode()**, see "Porting Display Mode Initialization Calls with auxInitDisplayMode()" on page 96. For information on **glXChooseVisual()**, see the glXIntro and **glXChooseVisual()** reference pages and refer to Chapter 5.)

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 **glReadPixels()** to read from the depth buffer, massage the results as necessary, and then draw the resulting data to an auxiliary buffer. Select this auxiliary buffer with **glReadBuffer()**, and use **glAccum()** to accumulate from that buffer. (Note that this procedure requires caution in converting among data types.)

Except as noted above, porting accumulation buffer calls should be straightforward. Table 3-36 lists calls that affect the accumulation buffer.

Table 3-36 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()	auxInitDisplayMode() or glXChooseVisual()	specify number of bitplanes per color component in the accumulation buffer

Table 3-37 lists IRIS GL's **acbuf()** arguments along with the corresponding arguments to OpenGL's **glAccum()**.

Table 3-37 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 **auxInitDisplayMode()** or **glXChooseVisual()**. (To learn how to use **auxInitDisplayMode()**, see "Porting Display Mode Initialization Calls with auxInitDisplayMode()" on page 96. For information on **glXChooseVisual()**,

see the glXIntro and **glXChooseVisual()** reference pages and refer to Chapter 5.) Otherwise, porting should be straightforward. Table 3-38 lists calls that affect the stencil planes.

Table 3-38 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 nearly 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 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.

Since 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()**

With IRIS GL, you used the commands **makeobj()** and **closeobj()** to create display lists. With 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 3-39 lists the IRIS GL display list commands with the corresponding OpenGL commands.

Table 3-39 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()	set 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.

Edited Display Lists

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.This OpenGL code fragment shows how:

```
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 ();
```

A Sample Implementation of a Display List

An IRIS GL display list might look like this:

```
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);
```

The example above defines three display lists, one of which refers to the others in its definition. Translated to OpenGL, that code might look like this:

```
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 does not have tables of stored definitions—you can’t 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'll probably need to port lighting and materials code by hand, since the OpenGL calls differ substantially from the IRIS GL calls. The OpenGL API is much cleaner, however; it has separate calls for setting lights, light models, and materials.

Porting notes:

- 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. This might 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 this 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 the same 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 3-40 lists IRIS GL lighting and materials commands and the corresponding OpenGL commands.

Table 3-40 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 3-41 lists the various materials parameters you can set.

Table 3-41 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

Table 3-41 (continued) Material Definition Parameters

lmdef() index	glMaterial() parameter	Default	Meaning
—	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'll need to replace **lmdef()** with several **glLight()** calls. Table 3-42 lists equivalent lighting model parameters.

Table 3-42 Lighting Model Parameters

lmdef() index	glLightModel() Parameter	Default	Meaning
AMBIENT	GL_LIGHT_MODEL_A MBIENT	(0.2, 0.2, 0.2, 1.0)	ambient color of scene
ATTENUATION	—	—	see glLight()
LOCALVIEWER	GL_LIGHT_MODEL_LO CAL_VIEWER	GL_FALSE	viewer local (TRUE) or infinite (FALSE)
TWOSIDE	GL_LIGHT_MODEL_T WO_SIDE	GL_FALSE	use two-sided lighting when TRUE

When the first argument of **lmdef()** is DEFLIGHT, the equivalent OpenGL call is **glLight()**. Table 3-43 lists equivalent lighting parameters.

Table 3-43 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

Here's an OpenGL code fragment that demonstrates some OpenGL lighting and material calls, including two-sided lighting:

```
/* 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

A single IRIS GL call might be replaced with two or more OpenGL calls, so you'll definitely need to edit the *toogl* output for these calls. You might need to restructure your code, or use more variables than you did before.

Porting notes:

- OpenGL 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.
- OpenGL doesn't automatically generate mipmaps for you—if you're using mipmaps, you'll need to call **gluBuild2DMipmaps()** first.
- You need to use **glEnable()** and **glDisable()** to turn texturing capabilities on and off. See the reference page for details.
- 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)

Table 3-44 lists the general OpenGL equivalents to IRIS GL texture calls.

Table 3-44 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

You'll probably want to look at the *OpenGL Programming Guide* to get details on how textures work in OpenGL, but here are a few brief, 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 pages 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 3-45 lists the IRIS GL texture environment options and their OpenGL equivalents.

Table 3-45 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()**. So 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 3-46 lists the IRIS GL texture parameters with their OpenGL equivalents. For more detailed information on OpenGL texture parameters, refer to the **glTexParameter()** reference page.

Table 3-46 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 3-47 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 might need to wait for RealityEngine extensions to the 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 3-47 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 need to call **glEnable()** and **glDisable()** to turn coordinate generation on and off. Table 3-48 lists the equivalents for texture coordinate names.

Table 3-48 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 3-49 lists texture generation mode and plane names.

Table 3-49 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);
```

Texturing in OpenGL: An Example

Here's an example of a complete OpenGL program demonstrating texture mapping:

```
/*  checker2.c
 *  This program texture maps a checkerboard image onto
 *  two rectangles.  This program repeats the texture, if
 *  the texture coordinates fall outside 0.0 and 1.0.
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

/*      Create checkerboard texture     */
#define checkImageWidth 64
#define checkImageHeight 64
GLubyte checkImage[checkImageWidth][checkImageHeight][3];

void makeCheckImage(void)
{
    int i, j, r, c;

    for (i = 0; i < checkImageWidth; i++) {
        for (j = 0; j < checkImageHeight; j++) {
            c = ((((i&0x8)==0)^((j&0x8))==0))*255;
            checkImage[i][j][0] = (GLubyte) c;
            checkImage[i][j][1] = (GLubyte) c;
            checkImage[i][j][2] = (GLubyte) c;
        }
    }
}

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);

    makeCheckImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, checkImageWidth,
        checkImageHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
        &checkImage[0][0][0]);
```

```
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                    GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_FLAT);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-2.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(-2.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(0.0, 1.0, 0.0);
    glTexCoord2f(3.0, 0.0); glVertex3f(0.0, -1.0, 0.0);

    glTexCoord2f(0.0, 0.0); glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(0.0, 3.0); glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(3.0, 3.0); glVertex3f(2.41421, 1.0,
                                       -1.41421);
    glTexCoord2f(3.0, 0.0); glVertex3f(2.41421, -1.0,
                                       -1.41421);
    glEnd();
    glFlush();
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h, 1.0,
                   30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}
```

```
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Porting Picking Calls

All the IRIS GL picking calls have OpenGL equivalents, with the exception of **clearhitcode()**. Table 3-50 lists the IRIS GL picking calls and their OpenGL counterparts.

Table 3-50 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*.

Here's an example of an OpenGL program that demonstrates picking:

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

#define SELECT 1
#define RENDER 2

void drawLine(int mode)
{
    float vec1[3] = {30.0, 30.0, 0.0};
    float vec2[3] = {50.0, 60.0, 0.0};
    float vec3[3] = {70.0, 40.0, 0.0};

    if (mode == SELECT)
      loadname(1);
    else
      cpack (0xFFFFFF);
    bgnline ();
      v3f (vec1);
      v3f (vec2);
    endline ();

    if (mode == SELECT)
      loadname(2);
    else
      cpack (0xFFFFFF);
    bgnline ();
      v3f (vec2);
      v3f (vec3);
    endline ();
}

#define BUFSIZE 50

void printhits(short buffer[], long hits)
{
    int indx, items, h, i;
    char str[20];

    printf("%ld hit", hits);

    if (hits != 1)
      printf("s");
```

```
    if (hits > 0)
      printf(": ");
    indx = 0;
    for (h = 0; h < hits; h++) {
      items = buffer[indx++];
      printf("(");
      for (i = 0; i < items; i++) {
          if (i != 0)
            printf(" ");
          printf("%d", buffer[indx++]);
       }
      printf(") ");
    }
    printf("\n");
}

main()
{
    Device dev;
    short val;
    long hits;
    long xsize, ysize;
    short buffer[BUFSIZE];
    Boolean run;

    prefsize (100, 100);
    winopen ("pickline");
    RGBmode ();
    gconfig ();
    getsize(&xsize, &ysize);
    mmode(MVIEWING);
    ortho2(0.0, 100.0, 0.0, 100.0);

    cpack(0);
    clear();
    qdevice(LEFTMOUSE);
    qdevice(ESCKEY);

    drawLine(RENDER);
    run = TRUE;

    picksize (5,5);
    while (run) {
      dev = qread(&val);
```

```
      if (val == 0) {                /* on upstroke */
        switch (dev) {
        case LEFTMOUSE:
          pushmatrix ();
            pick(buffer, BUFSIZE);
            ortho2(0.0, 100.0, 0.0, 100.0);
            drawLine(SELECT);/* no actual drawing happens */
            hits = endpick(buffer);
          popmatrix ();
          printhits(buffer, hits);
          break;

        case ESCKEY:
          run = FALSE;
          break;
        }
    }
    }
    gexit();
    return 0;
}
```

Here's how you could handle picking in OpenGL. This example uses auxiliary library calls for windowing and event handling.

```
/*
 *  pickline.c
 *  This code demonstrates picking.  Press the left mouse
 *  button to enter picking mode.  You get two hits if you
 *  click the intersection of the lines.
 */
#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

void drawLine(GLenum mode)
{
    if (mode == GL_SELECT)
      glLoadName (1);
    glBegin (GL_LINES);
      glColor3f (1.0, 1.0, 1.0);
      glVertex3f (30.0, 30.0, 0.0);
      glVertex3f (50.0, 60.0, 0.0);
    glEnd ();
```

```
    if (mode == GL_SELECT)
      glLoadName (2);
    glBegin (GL_LINES);
      glColor3f (1.0, 1.0, 1.0);
      glVertex3f (50.0, 60.0, 0.0);
      glVertex3f (70.0, 40.0, 0.0);
    glEnd ();
}

void printHits (GLint hits, GLuint buffer[])
{
    unsigned int i, j, names, ii, jj;
    unsigned int *ptr;

    printf ("hits = %d\n", hits);
    ptr = (unsigned int *) buffer;
    for (i = 0; i < hits; i++) {    /*  for each hit  */
      names = *ptr;
      printf (" number of names for this hit = %d\n", names);
      ptr++;
      printf ("  z1 is %u;", *ptr); ptr++;
      printf (" z2 is %u\n", *ptr); ptr++;
      printf ("   names are ");
      for (j = 0; j < names; j++) {    /*  for each name */
        printf ("%d ", *ptr);
        ptr++;
      }
      printf ("\n");
    }
}

#define BUFSIZE 512

void pickLine(AUX_EVENTREC *event)
{
    GLuint selectBuf[BUFSIZE];
    GLint hits;
    GLint viewport[4];
    int x, y;

    x = event->data[AUX_MOUSEX];
    y = event->data[AUX_MOUSEY];
    glGetIntegerv (GL_VIEWPORT, viewport);
```

```
    glSelectBuffer (BUFSIZE, selectBuf);
    (void) glRenderMode (GL_SELECT);

    glInitNames();
    glPushName(-1);

    glPushMatrix ();
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPickMatrix (x, 100-y, 5, 5, viewport);
    gluOrtho2D (0.0, 100.0, 0.0, 100.0);
    drawLine (GL_SELECT);
    glPopMatrix ();
    glFlush ();

    hits = glRenderMode (GL_RENDER);
    printHits (hits, selectBuf);
}

void display(void)
{
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, 100.0, 0.0, 100.0);

    glClearColor (0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    drawLine (GL_RENDER);
    glEnd ();
    glFlush();
}

void main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 100, 100);
    auxInitWindow (argv[0]);
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, pickLine);
    auxMainLoop(display);
}
```

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, implementation to implementation. Table 3-51 lists IRIS GL and OpenGL feedback calls.

Table 3-51 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.*

Here's an example demonstrating OpenGL feedback:

```
/*
 * 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/gl.h>
#include <GL/glu.h>
#include "aux.h"

/*    Initialize lighting.
 */
void myinit(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 primitives.
 */
void drawGeometry (long 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 ();
}

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");
}

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);
 (void) glRenderMode (GL_FEEDBACK);
 drawGeometry (GL_FEEDBACK);

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

```
void main(int argc, char** argv)
{
 auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
 auxInitPosition (0, 0, 100, 100);
 auxInitWindow (argv[0]);
 myinit ();
 auxMainLoop(display);
}
```

Porting RealityEngine Graphics Features

Unfortunately, some IRIS GL features that were available only on systems with RealityEngine graphics are unavailable in OpenGL.

Table 3-52 lists the IRIS GL RealityEngine calls and their OpenGL counterparts.

Table 3-52 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
displacepolygon()	glPolygonOffsetEXT()	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

Table 3-52 (continued) RealityEngine Calls

IRIS GL Call	OpenGL Call	Notes
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()	not supported	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

Table 3-52 (continued) RealityEngine Calls

IRIS GL Call	OpenGL Call	Notes
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
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()	glTexSubImage2DEXT() 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

Table 3-52 (continued) RealityEngine Calls

IRIS GL Call	OpenGL Call	Notes
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 3-52. 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:

- 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.

- IRIS GL's 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.)

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()**)
- 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.")

Chapter 4

Using the Auxiliary Library to Replace Windowing and Event Handling Calls

The auxiliary library provides several simple routines for windowing, event handling, loading the color map, and drawing several three-dimensional objects.

If your program uses only simple window and event handling calls, you can probably replace them with auxiliary library calls, rather than converting immediately to a mixed model program using Xlib or X. Remember that the auxiliary library is limited in functionality—more complicated applications will almost certainly require you to switch to mixed model instead of the auxiliary library.

For more information on the auxiliary library or on the auxiliary library calls, refer to the *OpenGL Programming Guide*. (At the time of this writing, reference pages are not available for auxiliary library routines.)

When using the auxiliary library, remember to include the auxiliary library header file:

```
#include "aux.h"
```

Running a Program That Uses the Auxiliary Library

To run a program using the auxiliary library, you need to call **auxMainLoop()** from within your **main()** routine, passing it the name of the routine that redraws the objects in your scene. The example programs in “Windowing with the Auxiliary Library: Example Program” on page 98 and “Input Handling with the Auxiliary Library: Example Program” on page 103 demonstrate how to do this.

Note: auxMainLoop() never exits, so calls that are placed after **auxMainLoop()** in your program are never executed.

Windowing

The auxiliary library provides three windowing routines: **auxInitWindow()**, **auxInitDisplayMode()**, and **auxInitPosition()**. With these routines, you can open and initialize a window.

Note that if you want to do more complex windowing, you'll need to use X calls. See Chapter 5, "Mixed-Model Programming," for more information. Also note that some IRIS GL window-related functions, notably **mswapbuffers()** and **swapinterval()**, have no real equivalents in OpenGL; these will probably be implemented in future OpenGL extensions.

Replacing prefposition() with auxInitPosition()

You can replace **prefposition()** with **auxInitPosition()**. The arguments don't correspond exactly, however. With **prefposition()** you specified the coordinates of opposite corners of the window. With **auxInitPosition()** you specify the screen coordinates (in pixels) of the upper left corner of the window, as well as the width and height of the window. The C specification for **auxInitPosition()** is:

```
void auxInitPosition( GLint x, GLint y, GLsizei width,
                      GLsizei height );
```

The default size for **auxInitPosition()** is a 100 x 100 pixel square. The default window position is at the upper left corner of the screen.

Porting Display Mode Initialization Calls with auxInitDisplayMode()

You can set the display mode of a window by selecting arguments to **auxInitDisplayMode()**. The C specification for **auxInitDisplayMode()** is:

```
void auxInitDisplayMode(GLbitfield mask)
```

The mask argument is a bitwise OR-ed combination of:

- AUX_RGBA or AUX_INDEX
- AUX_SINGLE or AUX_DOUBLE

and any of these buffer enabling flags:

- AUX_DEPTH, AUX_STENCIL, and AUX_ACCUM

So, for example, for a double-buffered, RGBA-mode window with a depth buffer, you would use:

```
auxInitDisplayMode( AUX_DOUBLE | AUX_RGBA | AUX_DEPTH );
```

The default setting for **auxInitDisplayMode()** is a single-buffered, color index window.

Table 4-1 IRIS GL lists display mode calls that can be either fully or partially replaced by **auxInitDisplayMode()**.

Table 4-1 **auxInitDisplayMode()** Arguments and IRIS GL Command Equivalents

IRIS GL Call	Corresponding auxInitDisplayMode() Argument	Specifies:
acsize()	AUX_ACCUM	accumulation-buffer mode
cmode()	AUX_INDEX	color-map mode
doublebuffer()	AUX_DOUBLE	double-buffer mode
RGBmode()	AUX_RGBA	RGBA mode
singlebuffer()	AUX_SINGLE	single-buffer mode
stensize()	AUX_STENCIL[a]	stencil-buffer mode
zbuffer(TRUE)	AUX_DEPTH	z-buffer mode

a. note that stencil() does more than specify stencil-buffer mode,—use **glStencilFunc()** to replace the rest of the stencil() functionality

Note: The auxiliary library doesn't provide—and you won't need—an equivalent for **gconfig()**.

auxInitDisplayMode() must be called before **auxInitWindow()**.

Replacing winopen() with auxInitWindow()

Use **auxInitWindow()** to replace **winopen()**. The C specification for **auxInitWindow()** is:

```
void auxInitWindow (GLbyte *titleString);
```

The specified string appears in the title bar of the window, and the escape key is bound to an exiting function that kills the window and exits the program. The default color for the background window is set to black for an RGBA window and to color index 0 for a color-index window.

Note: Remember that **auxInitDisplayMode()** must be called before **auxInitWindow()**.

Windowing with the Auxiliary Library: Example Program

Here's a simple program that uses the auxiliary library to initialize and open a window:

```
/*
 *  simple.c
 *  This program draws a white rectangle on a black
background.
 */
#include <GL/gl.h>
#include "aux.h"

int main(int argc, char** argv)
{
        auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
        auxInitPosition (0, 0, 500, 500);
        auxInitWindow (argv[0]);

        glClearColor (0.0, 0.0, 0.0, 0.0);
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0, 1.0, 1.0);
        glMatrixMode (GL_PROJECTION);
        glLoadIdentity ();
        glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
        glBegin(GL_POLYGON);
                glVertex2f(-0.5, -0.5);
                glVertex2f(-0.5, 0.5);
```

```
                glVertex2f(0.5, 0.5);
                glVertex2f(0.5, -0.5);
        glEnd();
        glFlush();
        sleep (10);
}
```

Event Handling: Replacing qdevice(), qtest(), and qread()

If you used the queuing method of event handling, you can use the auxiliary library to replace simple IRIS GL event handling calls. If you used polling calls, such as **getvaluator()** and **getbutton()**, you can't exactly replace that functionality with the auxiliary library. And, in general, for anything particularly complicated, you should probably invest the extra effort and translate your program to mixed model.

The auxiliary library's method of event handling is somewhat different from what you're used to in IRIS GL. With the auxiliary library, you structure your application's event handling to use callback functions. In general, you first open a window and register callback routines for specific events. Then you create a main loop without an exit. In the main loop, if an event occurs, its registered callback functions are executed. Upon completion of the callback functions, flow of control is returned to the main loop.

After a window is created, but before you enter the main loop, register callback functions using these three auxiliary library event handling routines: **auxReshapeFunc()**, **auxKeyFunc()**, and **auxMouseFunc()**.

Each of these three routines takes over some part of what you used to do with **qdevice()**, **qtest()**, and **qread()**, so read each section for details.

Handling Redraw Events

This section explains how to use **auxReshapeFunc()** to specify a function to be called whenever the window is resized, moved, or exposed.

In IRIS GL, you might have done something like this:

```
gid = winopen("MyProgram"); /*initialization*/
```

```
qenter(REDRAW,gid);
dev = qread (&value);
if ( dev == REDRAW ) {
       reshapeviewport();
       MySceneRedraw();
}
```

Using the auxiliary library, you specify a function that will be called automatically whenever you get a redraw event. Here's an example of a redraw function:

```
void myReshape(GLsizei w, GLsizei 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*/

}
```

Once you've written your redraw function, like the **myReshape()** in the example above, you have **auxReshapeFunc()** call it from the main loop. In this example, the main loop might look something like this:

```
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 400, 400);
    auxInitWindow (argv[0]);
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Here's the C specification for **auxReshapeFunc()**:

```
void auxReshapeFunc( void (*function) (GLsizei, GLsizei));
```

The argument function is a pointer to a function that expects two arguments: the new width and height of the window. Typically, the function calls

glViewport() so that the display is clipped to the new size, and it redefines the projection matrix so that the aspect ratio of the projected image matches the viewport, avoiding aspect ratio distortion.

If you don't call **auxReshapeFunc()**, a default reshape function is called, which assumes a two-dimensional orthographic projection. With the auxiliary library, the window is automatically redrawn after every reshaping event.

Handling Keyboard Input

With IRIS GL, you might do something like this:

```
qdevice (AKEY); /*initialization*/
```

```
dev = qread (&value);
if ((device == AKEY) && (val == 1))
    /* toggle autopilot mode */
    autopilot = ! autopilot;
```

In the auxiliary library, you use **auxKeyFunc()** to specify a function that will be called automatically whenever the A key is pressed. So, for this example, before entering the main loop of your program, you could define a function called **MyAutopilot()** to toggle the autopilot mode for you, then in the main loop of your program, you could call:

```
auxKeyFunc(AUX_a, MyAutopilot);
```

Then the auxiliary library will call MyAutopilot whenever the A key is pressed.

The C specification for **auxKeyFunc()** is:

```
void auxKeyFunc(GLint key, void (*function) (void) );
```

Specifies the function that is called when the key indicated by *key* is pressed. Use one of the auxiliary library constants for *key*:

- AUX_A through AUX_Z for the upper case alphabet
- AUX_a through AUX_z for the lower case alphabet
- AUX_0 through AUX_9 for the digits

- AUX_LEFT, AUX_RIGHT, AUX_UP, and AUX_DOWN for the arrow keys
- AUX_ESCAPE, AUX_SPACE, and AUX_RETURN for the escape key, the spacebar, and the return key, respectively

With the auxiliary library, the window is automatically redrawn after every processed key event, although in a more sophisticated application, you might want to wait for several events to be completed before drawing.

Handling Mouse Events

The C specification for **auxMouseFunc()** is:

```
void auxMouseFunc(GLint button, GLint mode,
                  void(*function)(AUX_EVENTREC *) );
```

auxMouseFunc() specifies the function that is called when the mouse button indicated by *button* enters the mode defined by *mode*. Here are the possible *button* arguments:

- AUX_LEFTBUTTON
- AUX_RIGHTBUTTON
- AUX_MIDDLEBUTTON

Here are the possible *mode* arguments:

- AUX_MOUSEDOWN
- AUX_MOUSEUP
- AUX_MOUSELOC

The function argument must take one argument that is a pointer to a structure of type AUX_EVENTREC. The **auxMouseFunc()** routine allocates memory for the structure. “Input Handling with the Auxiliary Library: Example Program” on page 103 contains a complete program using **auxMouseFunc()**.

With IRIS GL, you might do something like this:

```
qdevice (LEFTMOUSE); /*initialization*/
tie(LEFTMOUSE, MOUSEX, MOUSEY);
```

```
...
dev = qread (&value);

if (dev == LEFTMOUSE)
    {
        if (value)
        {
            qread(&x);
            qread(&y);
            qdevice(MOUSEX);
            qdevice(MOUSEY);
        }
        else
```

In the auxiliary library, you use **auxMouseFunc()** to specify functions that will be called automatically whenever the left mouse button is pressed or released. To replace the IRIS GL code above, you might define a function like this:

```
void MyFunction (AUX_EVENTREC *event ) {
    GLint x, y;
    x = event -> data[AUX_MOUSEX];
    y = event -> data[AUX_MOUSEY];
    ...
}
```

Then in your main loop you would call **auxMouseFunc()** like this:

```
auxMouseFunc(AUX_LEFTMOUSE, AUX_MOUSEDOWN, MyFunction);
```

For more information on **auxMouseFunc()**, see the *OpenGL Programming Guide*. You might also want to look at the *aux.h* include file for more information on defined constants, structure types, and so on.

Input Handling with the Auxiliary Library: Example Program

This program uses the auxiliary library for simple windowing and event handling. For another program example using the auxiliary library, see Appendix D, “Example OpenGL Program with the Auxiliary Library.”

```
/*  movelight.c
 *  This program demonstrates when to issue lighting and
```

```
 *  transformation commands to render a model with a light
 *  that is moved by a modeling transformation (rotate or
 *  translate).  The light position is reset after the
 *  modeling transformation is called.  The eye position
 *  does not change.

 *  A sphere is drawn using a gray material characteristic.
 *  A single light source illuminates the object.
 *
 *  Interaction:  pressing the left or middle mouse button
 *  alters the modeling transformation (x rotation) by 30
 *  degrees.
 *  The scene is then redrawn with the light in a new
 *  position.
 */

#include <GL/gl.h>
#include <GL/glu.h>
#include "aux.h"

static int spin = 0;

void movelight (AUX_EVENTREC *event)
{
    spin = (spin + 30) % 360;
}

void myinit (void)
{
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
```

```
    glDepthFunc(GL_LEQUAL);
    glEnable(GL_DEPTH_TEST);
}

/*  Here is where the light position is reset
 *  after the modeling transformation (glRotated)
 *  is called. This places the light at a new position
 *  in world coordinates. The cube represents the
 *  position of the light.
 */
void display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };

    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glTranslatef (0.0, 0.0, -5.0);

    glPushMatrix ();
    glRotated ((GLdouble) spin, 1.0, 0.0, 0.0);
    glLightfv (GL_LIGHT0, GL_POSITION, position);

    glTranslated (0.0, 0.0, 1.5);
    glDisable (GL_LIGHTING);
    glColor3f (0.0, 1.0, 1.0);
    auxWireCube (0.1);
    glEnable (GL_LIGHTING);
    glPopMatrix ();

    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();
    glFlush ();
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
}

/*  Main Loop
 *  Open window with initial window size, title bar,
```

```
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA | AUX_DEPTH);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit();
    auxMouseFunc (AUX_LEFTBUTTON, AUX_MOUSEDOWN, movelight);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Managing Background Events

You can specify a function to be executed if no other events are pending with **auxIdleFunc()**. Here's the C specification:

```
void auxIdleFunc(void (*func(void));
```

This routine takes a pointer to a function as its only argument. Pass in zero to disable the execution of the function.

Using Color-Index Mode

Loading a color map depends completely on the windowing system, so OpenGL doesn't provide any routines for this task. You can use the auxiliary library routine, **auxSetOneColor()**, instead of **mapcolor()**. Like **mapcolor()**, **auxSetOneColor()** takes four arguments: an index into the color map, and red, green, and blue values for intensity values of each of those colors.

The main thing to remember when you're switching from **mapcolor()** to **auxSetOneColor()** is that the red, green, and blue values are now normalized to lie in the range [0.0, 1.0]. So, for color values, **mapcolor()** took (short) integers between 0 and 255 and **auxSetOneColor()** takes floats between 0.0 and 1.0.

Here's an example of how you could use **auxSetOneColor()** to load values into the colormap:

```
for (i = 0; i < 32; i++) {
    auxSetOneColor (16 + i, 1.0 * (i/32.0),
                    0.0, 1.0 *(i/32.0));
    auxSetOneColor (48 + i, 1.0, 1.0 * (i/32.0), 1.0);
}
```

For a more complete example, see the program in "Input Handling with the Auxiliary Library: Example Program" on page 103

Here's the C specification for **auxSetOneColor()**:

```
auxSetOneColor (GLint index, GLfloat red, GLfloat green,
                GLfloat blue );
```

Note: As explained in "Porting Display Mode Initialization Calls with auxInitDisplayMode()" on page 96 you replace the initialization calls to **cmode()** and **gconfig()** by calling **auxInitDisplayMode()** with the argument AUX_INDEX.

Other Auxiliary Library Routines

The auxiliary library provides routines for drawing 3-D objects, such as cylinders, spheres, cubes, teapots, and more. The *OpenGL Programming Guide* discusses these routines in detail.

Chapter 5

Mixed-Model Programming

This chapter provides some preliminary information about OpenGL programming in the X environment. This chapter focuses specifically on information relevant to translating IRIS GL programs into mixed-model OpenGL / X programs—it does not provide a tutorial on Xt and IRIS IM. For more information on the relevant features of Xt and IRIS IM, consult the OSF / Motif series, and Digital's *X Window System Toolkit: The Complete Programmer's Guide and Specification*, or O'Reilly's Vols. 4 & 5 on X Toolkit Intrinsics.

What Is a Mixed-Model Program?

A mixed-model program, in the context of this manual, is an X program that creates one or more subwindows that use OpenGL for rendering. A mixed-model program allows full access to the capabilities of X by completely removing OpenGL from any feature governed by the X server, giving the programmer direct control of all the areas governed by the X server. In a mixed-model program, the X part of the code manages all of the event handling, window control, and menus. You also use X to handle color maps and fonts.

Note: You can't create mixed-model programs that go only halfway. Your mixed-model program must use X for all window-system-related services.

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

With IRIS GL, instead of creating a mixed-model program, you could use IRIS GL event and window management routines, such as **winopen()** or **qread()**—these routines would access the X Window System for you. Silicon Graphics does provide a very limited Auxiliary Library, which provides

some windowing and event handling routines, but if these are not sufficient for your purposes, you'll need to translate your IRIS GL code to a mixed-model program.

Porting IRIS GL Mixed-Model Programs

IRIS GL provided some mixed-model support, allowing IRIS GL programs to draw in X11 windows, with routines such as **GLXgetconfig()**, **GLXlink()**, **GLXunlink()**, and **GLXwinset()**. These functions don't have exact equivalents in OpenGL, but see Appendix A for approximate equivalents.

The naming conventions for X-related functions may be somewhat 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, confusingly, collectively referred to as "GLX." Note, too, that **GLXgetconfig()** (an IRIS GL mixed-model routine) is not at all the same function as **glXGetConfig()** (an OpenGL mixed-model 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 Mixed-Model Programming

When writing a mixed-model program, you have two choices: you can use the Xt toolkit and a widget set, such as IRIS IM, or you can write your program in Xlib and GL using special GLX commands.

The first method, using Xt and a widget set, is much easier and is more commonly used by mixed-model developers. This is the method recommended by Silicon Graphics—particularly for programmers with little or no previous experience with X.

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 mixed-model programming with Xt. "Porting to Mixed-Model Using Xt and a Widget Set" on page 114 explains how to convert your IRIS GL program to an OpenGL/X mixed-model program using Xt, the IRIS Widget Library, and the GL widget, GLwDrawingArea (Silicon Graphics also provides an IRIS IM version of GLwDrawingArea, called GLwMDrawingArea). Sample programs demonstrate these concepts.

Using Xlib

If you prefer to create a mixed-model program in Xlib, without using Xt, refer to the recommended references on X programming, and use the GLX routines described in the *OpenGL Reference Manual* (start with the glXIntro reference page). "Mixed-Model Programming Using Xlib and OpenGL GLX Commands" on page 122 provides more information on this mixed-model programming method 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.

Some General Hints on Mixed-Model Programming

This section contains a few brief hints for OpenGL/X mixed-model programmers. This list is not comprehensive—it just describes a few important things to watch out for.

You Can't Change Window Depth and Display Mode

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

Installing Color Maps

It's a good idea to call **XSetWMColormapWindows()** in your mixed-model program—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

The OpenGL contains no equivalents for the IRIS GL text-handling calls and Font Manager calls. To obtain full text- and font-handling facilities, use the OpenGL/X call **glXUseXFont()** with display lists to get some text-display capabilities. This section gives you an example.

To use display lists to do X bitmap fonts, your code 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 Helvetica 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.

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

GLuint base;

void makeRasterFont(void)
{
    XFontStruct *fontInfo;
    Font id;
    unsigned int first, last;
    Display *xdisplay;

    xdisplay = auxIdentifyXDisplay ();
    fontInfo = XLoadQueryFont(xdisplay,
            "-adobe-helvetica-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)
{
    glPushAttrib (GL_LIST_BIT);
    glListBase(base);
    glCallLists(strlen(s), GL_UNSIGNED_BYTE,
                (unsigned char *)s);
    glPopAttrib ();
}

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 ();
}
```

Porting to Mixed-Model Using Xt and a Widget Set

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

When mixing the GL with Xt, IRIS IM, or Athena widgets, you can use the Silicon Graphics mixed-model GLwDrawingArea widget, which simplifies

mixed-model 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.

What You Need to Know About Xt and IRIS IM

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

What is Xt?

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

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.

What Is IRIS IM?

IRIS IM is 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 the 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.

You Don’t Have to Use IRIS IM

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

About the GLwDrawingArea Widget

Combining OpenGL and Motif is made easier by a specially supplied OpenGL drawing area widget, GLwDrawingArea. Use the GLwDrawingArea widget when creating a mixed-model program using Xt. The GLwDrawingArea widget is similar to a normal widget, but it sets up a configuration for GL drawing, as well as providing resources and callbacks that are useful to the GL 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 with any widget set that is 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.
- GLwMDrawingArea has an IRIS IM style creation function, **GlxCreateMDrawingArea()**, in addition to allowing creation of the widget directly through Xt.

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

What You Need to Replace with X

You need to replace any GL code that handled anything controlled by the X Window System—this will mainly include your windowing and event handling code. One way to do this is to run *toogl* and then search through the output for the *toogl* warnings, which it marks "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 below, 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.)

```
/* 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 is a good idea to always call **GlxDrawingAreaMakeCurrent()** to set the current widget. In the previous program, **GlxDrawingAreaMakeCurrent()** is called from the callback functions.

In the program example shown above, a wire frame cylinder is drawn 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 does not require steady input events. Appendix E, "Example Mixed-Model Program with WorkProc," contains an example mixed-model program using Xt and WorkProc.

Other Information Sources for Mixed-Model Programming

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, refer to 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 also 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.

Mixed-Model Programming Using Xlib and OpenGL GLX Commands

This method of mixed-model programming is more difficult 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 mixed-model programming 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.

In brief, to port your OpenGL code to a mixed-model program using Xlib and GLX calls, follow these steps:

1. Add the necessary include files to your program. (See "Header Files" on page 15 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()**.

Code Example: Opening a Window with OpenGL GLX

The following example is a simple way of following those steps.You can find a version of this code in the glXIntro reference page. This sample is more heavily commented and contains some additional examples:

```
#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);
}
```

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);
}
```

A Sample X Event Loop

Here's a simple example of a mixed-model 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 the introductory section of this guide, "About This Guide."

Table A-1 IRIS GL Commands and Their OpenGL Equivalents

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
acbuf()	glAccum()	"Accumulation Buffer Calls" on page 61
acsize()	glXChooseVisual()	"Accumulation Buffer Calls" on page 61
addtopup()	use X or IRIS IM for menus	Chapter 5, 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/aux Equivalent	Where Discussed
arc(), arcf()	gluPartialDisk()[a]	"Editing toogl Output: An Example" on page 14 and "Arcs and Circles" on page 41
backbuffer()	glDrawBuffer(GL_BACK)	
backface()	glCullFace(GL_BACK)	
bbox2()	not supported	"Porting bbox2() Calls" on page 64
bgnclosedline()	glBegin(GL_LINE_LOOP)	"bgn/end Commands" on page 31 and "Lines" on page 35
bgncurve()	gluBeginCurve()	"NURBS Curves" on page 52
bgnline()	glBegin(GL_LINE_STRIP)	"bgn/end Commands" on page 31 and "Lines" on page 35
bgnpoint()	glBegin(GL_POINTS)	"bgn/end Commands" on page 31 and "Points" on page 34
bgnpolygon()	glBegin(GL_POLYGON)	"bgn/end Commands" on page 31, "Polygons and Quadrilaterals" on page 36 and "Tessellated Polygons" on page 40
bgnqstrip()	glBegin(GL_QUAD_STRIP)	"bgn/end Commands" on page 31 and "Polygons and Quadrilaterals" on page 36
bgnsurface()	gluBeginSurface()	"NURBS Surfaces" on page 54

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
bgntmesh()	glBegin(GL_TRIANGLE_STRIP)	"bgn/end Commands" on page 31 and "Triangles" on page 40
bgntrim()	gluBeginTrim()	"Trimming Curves" on page 54
blankscreen()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
blanktime()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
blendcolor()	glBlendColorEXT()	"Porting RealityEngine Graphics Features" on page 90
blendfunction()	glBlendFunc()	"Blending" on page 58
blink()	use auxiliary library or X for color maps	"Using Color-Index Mode" on page 106 or Chapter 5 and glXIntro reference page
blkqread()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
c3*(), c4*()	glColor*()	"Porting Color, Shading, and Writemask Commands" on page 43
callfunc()	not supported	"Porting Display Lists" on page 63
callobj()	glCallList()	"bgn/end Commands" on page 31 and "Porting Display Lists" on page 63

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
charstr()	glCallLists()a	"Fonts and Strings" on page 112
chunksize()	not needed	"Porting Display Lists" on page 63
circ(), circf()	gluDisk()	"Arcs and Circles" on page 41
clear()	glClear(GL_COLOR_BUFFER_BIT)	"Windowing, Device, and Event Calls" on page 10 and "Porting Screen and Buffer Clearing Commands" on page 20
clearhitcode()	not supported	"Porting Picking Calls" on page 81
clipplane()	glClipPlane()	"Clipping Planes" on page 29
clkon()	XChangeKeyboardControl()	see X documentation
clkoff()	XChangeKeyboardControl()	see X documentation
closeobj()	glEndList()	"Porting Display Lists" on page 63
cmode()	auxInitDisplayMode() or glXChooseVisual()	"Using Color-Index Mode" on page 106 or Chapter 5 and glXIntro and glXChooseVisual() reference pages
cmov(), cmov2()	glRasterPos3()[a], glRasterPos2()[a]	"Porting Pixel Operations" on page 46
color(), colorf()	glIndex*()	"bgn / end Commands" on page 31 and "Porting Color, Shading, and Writemask Commands" on page 43

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
compactify()	not needed	"Porting Color, Shading, and Writemask Commands" on page 43
concave()	gluBeginPolygon()[a]	
convolve()	glConvolutionFilter2DEXT(), glSeparableFilter2DEXT(), glConvolutionParameterEXT(), glPixelTransfer()	"Porting RealityEngine Graphics Features" on page 90
cpack()	glColor*()[a]	"bgn/end Commands" on page 31 and "Porting Color, Shading, and Writemask Commands" on page 43
crv()	not supported	"Porting Curve and Surface Commands" on page 51
crvn()	not supported	"Porting Curve and Surface Commands" on page 51
curorigin()	use X for cursors	Chapter 5, glXIntro reference page, X documentation
cursoff()	use X for cursors	Chapter 5, glXIntro reference page, X documentation
curson()	use X for cursors	Chapter 5, glXIntro reference page, X documentation
curstype()	use X for cursors	Chapter 5, glXIntro reference page, X documentation
curvebasis()	glMap1()	"Porting Curve and Surface Commands" on page 51

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
curveit()	glEvalMesh1()	"Porting Curve and Surface Commands" on page 51
curveprecision()	not supported	"Porting Curve and Surface Commands" on page 51
cyclemap()	use auxiliary library or X for color maps	"Using Color-Index Mode" on page 106 or Chapter 5 and glXIntro reference page
czclear()	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)	"Porting Screen and Buffer Clearing Commands" on page 20
dbtext()	not supported	Dial and Button Box documentation
defbasis()	glMap1()	"Porting Curve and Surface Commands" on page 51
defcursor()	use X for cursors	Chapter 5, glXIntro reference page, X documentation
deflinestyle()	glLineStipple()	"Lines" on page 35 and "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67
defpattern()	glPolygonStipple()	"Polygons and Quadrilaterals" on page 36 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/aux Equivalent	Where Discussed
defpup()	use X for menus	Chapter 5, glXIntro reference page, X documentation
defrasterfont()	glXUseXFont()[a]	"Fonts and Strings" on page 112
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()	"Porting RealityEngine Graphics Features" on page 90
dither()	glEnable(GL_DITHER)	"Porting Color, Shading, and Writemask Commands" on page 43
dopup()	use X for menus	Chapter 5, glXIntro reference page, X documentation
doublebuffer()	auxInitDisplayMode() or glXChooseVisual()	"Porting Display Mode Initialization Calls with auxInitDisplayMode()" on page 96 or Chapter 5 and glXIntro reference page

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
draw()	glBegin(GL_LINES)[a]	"Porting Commands that Required Current Graphics Position" on page 19 and "Lines" on page 35
drawmode()	glXMakeCurrent()[a]	
editobj()	not supported	"Porting Display Lists" on page 63
endclosedline()	glEnd()	"bgn/end Commands" on page 31 and "Lines" on page 35
endcurve()	gluEndCurve()	"Porting Curve and Surface Commands" on page 51
endfeedback()	glRenderMode(GL_RENDER)	"Porting Feedback Calls" on page 87
endfullscreen()	not supported	
endline()	glEnd()	"bgn/end Commands" on page 31
endpick()	glRenderMode(GL_RENDER)	"Porting Picking Calls" on page 81
endpoint()	glEnd()	"bgn/end Commands" on page 31 and "Points" on page 34
endpolygon()	glEnd()	"bgn/end Commands" on page 31 and "Polygons and Quadrilaterals" on page 36
endpupmode()	use X for menus	Chapter 5, glXIntro reference page, X documentation

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
endqstrip()	glEnd()	"bgn/end Commands" on page 31 and "Polygons and Quadrilaterals" on page 36
endselect()	glRenderMode(GL_RENDER)	"Porting Picking Calls" on page 81
endsurface()	gluEndSurface()	"NURBS Surfaces" on page 54
endtmesh()	glEnd()	"bgn/end Commands" on page 31 and "Triangles" on page 40
endtrim()	gluEndTrim()	"Trimming Curves" on page 54
fbsubtexload()	not supported	"Porting RealityEngine Graphics Features" on page 90
feedback()	glFeedbackBuffer()	"Porting Feedback Calls" on page 87
finish()	glFinish()	
fogvertex()	glFog()	"Porting Depth Cueing and Fog Commands" on page 48
font()	see glListBase()	
foreground()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
freepup()	use X for menus	Chapter 5, glXIntro reference page, X documentation
frontbuffer()	glDrawBuffer(GL_FRONT)	

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
frontface()	see glCullFace()	
fudge()	use auxiliary library or X for windowing	
fullscrn()	not supported	
gammaramp()	use auxiliary library or X for color maps	"Using Color-Index Mode" on page 106 or Chapter 5 and glXIntro reference page
gbegin()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
gconfig()	no equivalent (not needed)	"Porting Display Mode Initialization Calls with auxInitDisplayMode()" on page 96 or Chapter 5 and glXIntro reference page
genobj()	glGenLists()	"Porting Display Lists" on page 63
gentag()	not supported	"Stencil Plane Calls" on page 62
getbackface()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18
getbuffer()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18
getbutton()	use auxiliary library or X for windowing	"Porting IRIS GL 'Get' Commands" on page 18, "Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5, and glXIntro reference page

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
getcmmode()	glXGetCurrentContext()	"Porting IRIS GL 'Get' Commands" on page 18, Chapter 5 and "Porting Color, Shading, and Writemask Commands" on page 43
getcolor()	glGet()	
getcpos()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18
getcursor()	not supported	"Porting IRIS GL 'Get' Commands" on page 18
getdcm()	glIsEnabled()	"Porting IRIS GL 'Get' Commands" on page 18 and "Porting Depth Cueing and Fog Commands" on page 48
getdepth()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18
getdescender()	use X for fonts	"Fonts and Strings" on page 112 and "Porting IRIS GL 'Get' Commands" on page 18
getdev()	not supported	"Porting IRIS GL 'Get' Commands" on page 18
getdisplaymode()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18
getdrawmode()	glXGetCurrentContext()	"Porting IRIS GL 'Get' Commands" on page 18
getfont()	use X for fonts	"Porting IRIS GL 'Get' Commands" on page 18 and "Fonts and Strings" on page 112

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
getgdesc()	glGet(), glXGetConfig(), glXGetCurrentContext(), glXGetCurrentDrawable()	"Porting IRIS GL 'Get' Commands" on page 18
getgpos()	not supported	"Porting Commands that Required Current Graphics Position" on page 19 and "Porting IRIS GL 'Get' Commands" on page 18
getheight()	use X for fonts	"Fonts and Strings" on page 112 and "Porting IRIS GL 'Get' Commands" on page 18
gethgram()	glGetHistogramEXT()	"Porting RealityEngine Graphics Features" on page 90
gethitcode()	not supported	"Porting Picking Calls" on page 81 and "Porting IRIS GL 'Get' Commands" on page 18
getlsbackup()	not supported	"Lines" on page 35 and "Porting IRIS GL 'Get' Commands" on page 18
getlsrepeat()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18 and "Lines" on page 35
getlstyle()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18 and "Lines" on page 35
getlwidth()	glGet()	"Porting IRIS GL 'Get' Commands" on page 18 and "Lines" on page 35

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
getmap(void)	not supported	"Porting IRIS GL 'Get' Commands" on page 18, Chapter 5 and "Porting Color, Shading, and Writemask Commands" on page 43
getmatrix()	glGet(GL_MODELVIEW_ MATRIX), glGet(GL_PROJECTION_ MATRIX)	"Porting IRIS GL 'Get' Commands" on page 18 and "Porting Matrix and Transformation Calls" on page 22
getmcolor()	not supported	"Porting IRIS GL 'Get' Commands" on page 18, "Porting Color, Shading, and Writemask Commands" on page 43, "Using Color-Index Mode" on page 106, or Chapter 5 and glXIntro reference page
getminmax()	glGetMinmaxEXT()	"Porting RealityEngine Graphics Features" on page 90
getmmode()	glGet(GL_MATRIX_MODE)	"Porting IRIS GL 'Get' Commands" on page 18 and "Porting Matrix and Transformation Calls" on page 22
getmonitor()	not supported	"Porting IRIS GL 'Get' Commands" on page 18
getnurbsproperty()	gluGetNurbsProperty()	"Porting IRIS GL 'Get' Commands" on page 18

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
getopenobj()	not supported	"Porting Display Lists" on page 63 and "Porting IRIS GL 'Get' Commands" on page 18
getorigin()	use auxiliary library or X for windowing	"Porting IRIS GL 'Get' Commands" on page 18 and "Windowing" on page 96, or Chapter 5 and glXIntro reference page
getpattern()	glGetPolygonStipple()	"Porting IRIS GL 'Get' Commands" on page 18 and "Polygons and Quadrilaterals" on page 36
getplanes()	glGet(GL_RED_BITS), glGet(GL_GREEN_BITS), glGet(GL_BLUE_BITS)	"Porting IRIS GL 'Get' Commands" on page 18
getport()	use auxiliary library or X for windowing	"Porting IRIS GL 'Get' Commands" on page 18, "Windowing" on page 96 or Chapter 5 and glXIntro reference page
getresetls()	not supported	"Lines" on page 35 and "Porting IRIS GL 'Get' Commands" on page 18
getscrbox()	not supported	"Porting IRIS GL 'Get' Commands" on page 18 and "Viewports, Screenmasks, and Scrboxes" on page 28
getscrmask()	glGet(GL_SCISSOR_BOX)	"Porting IRIS GL 'Get' Commands" on page 18 and "Viewports, Screenmasks, and Scrboxes" on page 28

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
getshade()	glGet(GL_CURRENT_INDEX)	"Porting IRIS GL 'Get' Commands" on page 18
getsize()	use auxiliary library or X for windowing	"Porting IRIS GL 'Get' Commands" on page 18, "Windowing" on page 96 or Chapter 5 and glXIntro reference page
getsm()	glGet(GL_SHADE_MODEL)	"Porting IRIS GL 'Get' Commands" on page 18 and "Porting Color, Shading, and Writemask Commands" on page 43
getvaluator()	use auxiliary library or X for event handling	"Porting IRIS GL 'Get' Commands" on page 18, "Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
getvideo()	not supported	"Porting IRIS GL 'Get' Commands" on page 18
getviewport()	glGet(GL_VIEWPORT)	"Porting IRIS GL 'Get' Commands" on page 18 and "Viewports, Screenmasks, and Scrboxes" on page 28
getwritemask()	glGet(GL_INDEX_WRITEMASK)	"Porting IRIS GL 'Get' Commands" on page 18 and "Porting Color, Shading, and Writemask Commands" on page 43
getwscrn()	use auxiliary library or X for windowing	"Porting IRIS GL 'Get' Commands" on page 18 and "Windowing" on page 96 or Chapter 5 and glXIntro reference page

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
getzbuffer()	glIsEnabled(GL_DEPTH_TEST)	"Porting IRIS GL 'Get' Commands" on page 18
gexit()	use auxiliary library or X for windowing	
gflush()	glFlush()	
ginit()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
glcompat()	not supported	
GLXgetconfig()	roughly, glXChooseVisual()	Chapter 5 and glXIntro reference page
GLXlink()	combination of glXCreateContext() and glXMakeCurrent()	Chapter 5 and glXIntro reference page
GLXunlink()	glXMakeCurrent(display_name, None, NULL)	Chapter 5 and glXIntro reference page
GLXwinset()	roughly, glXMakeCurrent()	Chapter 5 and glXIntro reference page
greset()	not supported	"Porting greset()" on page 16
gRGBcolor()	glGet(GL_CURRENT_ RASTER_COLOR)	"Porting Color, Shading, and Writemask Commands" on page 43
gRGBcursor()	use X for cursors	Chapter 5, glXIntro reference page, X documentation
gRGBmask()	glGet(GL_COLOR_WRITEMASK)	"Porting Color, Shading, and Writemask Commands" on page 43
gselect()	glSelectBuffer()	

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
gsync()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
gversion()	glGetString(GL_RENDERER)[a]	Chapter 5 and glXIntro reference page
hgram()	glHistogramEXT(), glResetHistogramEXT()	"Porting RealityEngine Graphics Features" on page 90
iconsize()	use X	see X documentation for XIconSize()
icontitle()	use X	see X documentation for XSetIconName()
ilbuffer()	not supported	"Porting RealityEngine Graphics Features" on page 90
ildraw()	not supported	"Porting RealityEngine Graphics Features" on page 90
imakebackground()	use the auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
initnames()	glInitNames()	
ismex()	not supported	"Porting IRIS GL 'Get' Commands" on page 18
isobj()	glIsList()	"Porting Display Lists" on page 63
isqueued()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
istag()	not supported	"Stencil Plane Calls" on page 62
istexloaded()	not supported	"Porting RealityEngine Graphics Features" on page 90
keepaspect()	use the auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
lampoff()	not supported	see X documentation for XChangeKeyboardControl()
lampon()	not supported	see X documentation for XChangeKeyboardControl()
leftbuffer()	glDrawBuffer(GL_LEFT)	"Porting RealityEngine Graphics Features" on page 90
linesmooth()	glEnable(GL_LINE_SMOOTH)	"Lines" on page 35 and "Antialiasing Calls" on page 60
linewidth()	glLineWidth()	"Lines" on page 35
linewidthf()	glLineWidth()	"Lines" on page 35
lmbind()	glEnable(GL_LIGHTING) glEnable(GL_LIGHTi)	"bgn / end Commands" on page 31, "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

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
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 22
loadname()	glLoadName()	"Porting Picking Calls" on page 81
logicop()	glLogicOp()	"Porting Pixel Operations" on page 46
lookat()	gluLookAt()[a]	"Porting Matrix and Transformation Calls" on page 22
lrectread()	glReadPixels()	"Porting Pixel Operations" on page 46
lrectwrite(])	glDrawPixels()	"Porting Pixel Operations" on page 46
lRGBrange()	not supported; see glFog()	"Porting Depth Cueing and Fog Commands" on page 48
lsbackup()	not supported	"Lines" on page 35
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()	"Lines" on page 35

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
makeobj()	glNewList()	"Porting Display Lists" on page 63
maketag()	not supported	"Stencil Plane Calls" on page 62
mapcolor()	auxSetOneColor() or XStoreColor()	"Using Color-Index Mode" on page 106 and Chapter 5
mapw()	gluProject()	"Porting Matrix and Transformation Calls" on page 22
maxsize()	use the auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
minmax()	glMinmaxEXT()	"Porting RealityEngine Graphics Features" on page 90
minsize()	use the auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
mmode()	glMatrixMode()	"Porting Matrix and Transformation Calls" on page 22
monobuffer()	superseded by selection of an appropriate GLX visual	glXChooseVisual() reference page and "Porting RealityEngine Graphics Features" on page 90
move()	not supported	"Porting Commands that Required Current Graphics Position" on page 19

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
msalpha()	glEnable(GL_SAMPLE_ALPHA_TO_MASK_SGIS), glEnable(GL_SAMPLE_ALPHA_TO_ONE_SGIS)	"Porting RealityEngine Graphics Features" on page 90
msmask()	glSampleMaskSGIS()	"Porting RealityEngine Graphics Features" on page 90
mspattern()	glSamplePatternSGIS()	"Porting RealityEngine Graphics Features" on page 90
mssize()	glXChooseVisual with attribute GLX_SAMPLE_BUFFERS_SGIS	"Porting RealityEngine Graphics Features" on page 90
mswapbuffers()	not supported	"Windowing" on page 96
multimap()	use the auxiliary library or X for color maps	"Porting Color, Shading, and Writemask Commands" on page 43, also "Using Color-Index Mode" on page 106 or Chapter 5 and glXIntro reference page
multisample()	glEnable(GL_MULTISAMPLE_SGIS)	"Porting RealityEngine Graphics Features" on page 90
multmatrix()	glMultMatrix()	
n3f()	glNormal3fv()	"bgn/end Commands" on page 31
newpup()	use X for menus	Chapter 5, glXIntro reference page, X documentation
newtag()	not supported	"Porting Display Lists" on page 63
nmode()	glEnable(GL_NORMALIZE)	

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
noborder()	use the auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
noise()	use the auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
noport()	use X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
normal()	glNormal3fv()	
nurbscurve()	gluNurbsCurve()[a]	"NURBS Curves" on page 52 and "Trimming Curves" on page 54
nurbssurface()	gluNurbsSurface()[a]	"NURBS Surfaces" on page 54
objdelete()	not supported	"Stencil Plane Calls" on page 62
objinsert()	not supported	"Stencil Plane Calls" on page 62
objreplace()	not supported	"Porting Display Lists" on page 63
onemap()	use the auxiliary library or X for color maps	"Porting Color, Shading, and Writemask Commands" on page 43 and "Using Color-Index Mode" on page 106 or Chapter 5 and glXIntro reference page
ortho()	glOrtho()	"Porting Matrix and Transformation Calls" on page 22

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
ortho2()	gluOrtho2D()	"Porting Matrix and Transformation Calls" on page 22
overlay()	use X	Chapter 5 and glXIntro reference page, also glXChooseVisual()
pagecolor()	not supported	
passthrough()	glPassThrough()	"Porting Feedback Calls" on page 87
patch()	glEvalMesh2()[a]	"Porting Curve and Surface Commands" on page 51
patchbasis()	glMap2()[a]	"Porting Curve and Surface Commands" on page 51
patchcurves()	glMap2()[a]	"Porting Curve and Surface Commands" on page 51
patchprecision()	not supported	"Porting Curve and Surface Commands" on page 51
pclos()	not supported; see glEnd()	"Porting Commands that Required Current Graphics Position" on page 19 and "Polygons and Quadrilaterals" on page 36
pdr()	not supported; see glVertex()	"Porting Commands that Required Current Graphics Position" on page 19 and "Polygons and Quadrilaterals" on page 36

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
perspective()	gluPerspective()	"Porting Matrix and Transformation Calls" on page 22
pick()	gluPickMatrix()[a] and glRenderMode(GL_SELECT)	"Porting Picking Calls" on page 81
picksize()	gluPickMatrix()	"Porting Matrix and Transformation Calls" on page 22 and "Porting Picking Calls" on page 81
pixelmap()	glPixelMap()	"Porting RealityEngine Graphics Features" on page 90
pixeltransfer()	glPixelTransfer()	"Porting RealityEngine Graphics Features" on page 90
pixmode()	glPixelTransfer() and glPixelStore()	"Porting Pixel Operations" on page 46
pmv()	not supported; see glBegin() and glVertex()	"Porting Commands that Required Current Graphics Position" on page 19 and "Polygons and Quadrilaterals" on page 36
pnt*()	glBegin(GL_POINTS)[a]	"Points" on page 34
pntsize(), pntsizef()	glPointSize()	"Points" on page 34
pntsmooth()	glEnable(GL_POINT_SMOOTH)	"Points" on page 34 and "Antialiasing Calls" on page 60
polarview()	not supported; see glRotate() and glTranslate()	"Porting Matrix and Transformation Calls" on page 22

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
polf()	not supported	"Polygons and Quadrilaterals" on page 36
poly()	not supported	"Polygons and Quadrilaterals" on page 36
polymode()	glPolygonMode()	"Polygons and Quadrilaterals" on page 36
polysmooth()	glEnable(GL_POLYGON_SMOOTH)	"Antialiasing Calls" on page 60
popattributes()	glPopAttrib()	
popmatrix()	glPopMatrix()	"Porting Matrix and Transformation Calls" on page 22
popname()	glPopName()	"Porting Picking Calls" on page 81
popviewport()	glPopAttrib()	"Viewports, Screenmasks, and Scrboxes" on page 28
prefposition()	use auxiliary library or X for windowing	"Replacing prefposition() with auxInitPosition()" on page 96 or Chapter 5 and glXIntro reference page
prefsize()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
pupmode()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
pushattributes()	glPushAttrib()	

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
pushmatrix()	glPushMatrix()	"Porting Matrix and Transformation Calls" on page 22
pushname()	glPushName()	"Porting Picking Calls" on page 81
pushviewport()	glPushAttrib(GL_VIEWPORT)	"Viewports, Screenmasks, and Scrboxes" on page 28
pwlcurve()	gluPWLCurve()	"Trimming Curves" on page 54
qcontrol()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
qdevice()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
qenter()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
qgetfd()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
qread()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
qreset()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
qtest()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
rcrv()	not supported	"Porting Curve and Surface Commands" on page 51
rcrvn()	not supported	"Porting Curve and Surface Commands" on page 51
rdr()	not supported	"Porting Commands that Required Current Graphics Position" on page 19
readcomponent()	glReadPixels() gives partial support; some readcomponent() features aren't supported	"Porting RealityEngine Graphics Features" on page 90
readdisplay()	not supported	
readpixels()	glReadPixels()	"Porting Pixel Operations" on page 46
readRGB()	not supported	"Porting Pixel Operations" on page 46
readsource()	glReadBuffer()	"Porting Pixel Operations" on page 46
rect(), rectf()	see glRect() and glPolygonMode()	"Polygons and Quadrilaterals" on page 36

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
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	"Lines" on page 35
reshapeviewport()	not supported	"Handling Redraw Events" on page 99 or Chapter 5 and glXIntro reference page
RGBcolor()	glColor()	"bgn/end Commands" on page 31 and "Porting Color, Shading, and Writemask Commands" on page 43
RGBcursor()	use X for cursors	Chapter 5, glXIntro reference page, X documentation
RGBmode()	use auxiliary library or X for windowing	"Porting Display Mode Initialization Calls with auxInitDisplayMode()" on page 96 or Chapter 5 and glXIntro reference page
RGBrange()	not supported	
RGBsize()	not supported, but the glXChooseVisual() function does some similar things	glXChooseVisual() reference page

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
RGBwritemask()	glColorMask()	"Porting Color, Shading, and Writemask Commands" on page 43
rightbuffer()	glDrawBuffer(GL_RIGHT)	"Porting RealityEngine Graphics Features" on page 90
ringbell()	not supported	see X documentation for XBell()
rmv()	not supported	"Porting Commands that Required Current Graphics Position" on page 19
rot()	glRotate()	"Porting Matrix and Transformation Calls" on page 22
rotate()	glRotate()	"Porting Matrix and Transformation Calls" on page 22
rpatch()	not supported	"Porting Curve and Surface Commands" on page 51
rpdr()	not supported	"Porting Commands that Required Current Graphics Position" on page 19 and "Polygons and Quadrilaterals" on page 36
rpmv()	not supported	"Porting Commands that Required Current Graphics Position" on page 19 and "Polygons and Quadrilaterals" on page 36

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
sbox(), sboxf()	glRect()[a]	"Polygons and Quadrilaterals" on page 36
scale()	glScale()	"Porting Matrix and Transformation Calls" on page 22
sclear()	glClear(GL_STENCIL_BUFFER_BIT)	"Porting Screen and Buffer Clearing Commands" on page 20 and "Stencil Plane Calls" on page 62
scrbox()	not supported	"Viewports, Screenmasks, and Scrboxes" on page 28
screenspace()	not supported	"Porting Matrix and Transformation Calls" on page 22
scrmask()	glScissor()	"Viewports, Screenmasks, and Scrboxes" on page 28
scrnattach()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
scrnselect()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
scrsubdivide()	not supported	
select()	glRenderMode()	"Porting Picking Calls" on page 81
setbell()	not supported	see X documentation for XChangeKeyboardControl()

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
setcursor()	use X for cursors	Chapter 5, glXIntro reference page and X documentation
setdblights()	not supported	dial and button box documentation
setdepth()	glDepthRange()[a]	
setlinestyle()	glLineStipple()	"Lines" on page 35 and "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67
setmap()	use auxiliary library or X for color maps	"Porting Color, Shading, and Writemask Commands" on page 43 and "Using Color-Index Mode" on page 106 or Chapter 5 and glXIntro reference page
setmonitor()	not supported	
setnurbsproperty()	gluNurbsProperty()	
setpattern()	glPolygonStipple()	"Polygons and Quadrilaterals" on page 36 and "Porting defs, binds, and sets: Replacing 'Tables' of Stored Definitions" on page 67
setpup()	use X for menus	Chapter 5, glXIntro reference page, X documentation
setvaluator()	use X for devices	Chapter 5, glXIntro reference page, X documentation
setvideo()	not supported	

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
shademodel()	glShadeModel()	"Porting Color, Shading, and Writemask Commands" on page 43
shaderange()	glFog()	
singlebuffer()	use auxiliary library or X for windowing	"Porting Display Mode Initialization Calls with auxInitDisplayMode()" on page 96 or Chapter 5 and glXIntro reference page
smoothline()	glEnable(GL_LINE_SMOOTH)	"Lines" on page 35
spclos()	not supported	"Polygons and Quadrilaterals" on page 36
splf()	not supported see glBegin()	"Polygons and Quadrilaterals" on page 36
stencil()	glStencilFunc(), glStencilOp()	"Stencil Plane Calls" on page 62
stensize()	glStencilMask()	"Stencil Plane Calls" on page 62
stepunit()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
stereobuffer()	superseded by selection of an appropriate GLX visual	glXChooseVisual() reference page and "Porting RealityEngine Graphics Features" on page 90
strwidth()	use X for fonts and strings	"Fonts and Strings" on page 112
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/aux Equivalent	Where Discussed
subtexload()	glTexSubImage2DEXT()	"Porting RealityEngine Graphics Features" on page 90
swapbuffers()	glXSwapBuffers()	glXIntro and glXSwapBuffers() reference pages
swapinterval()	not supported	"Windowing" on page 96
swaptmesh()	not supported; see glBegin(GL_TRIANGLE_FAN)	"Triangles" on page 40
swinopen()	use the auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
swritemask()	glStencilMask()	"Stencil Plane Calls" on page 62
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

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
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
texdef3d()	glTexImage3DEXT()	"Porting RealityEngine Graphics Features" on page 90
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 auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
tlutbind()	not supported	"Porting RealityEngine Graphics Features" on page 90
tlutdef()	not supported	"Porting RealityEngine Graphics Features" on page 90

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
tpoff()	not supported	
tpon()	not supported	
translate()	glTranslate()	"Porting Matrix and Transformation Calls" on page 22
underlay()	glXChooseVisual()	
unqdevice()	use auxiliary library or X for event handling	"Event Handling: Replacing qdevice(), qtest(), and qread()" on page 99 or Chapter 5 and glXIntro reference page
v2*(), v3*(), v4*()	glVertex*()	"The v() Commands" on page 31
videocmd()	not supported	
viewport()	glViewport()	"Viewports, Screenmasks, and Scrboxes" on page 28
winattach()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
winclose()	glXDestroyContext(), XCloseDisplay()	
winconstraints()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
windepth()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
window()	glFrustum()	"Porting Matrix and Transformation Calls" on page 22

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
winget()	glXGetCurrentContext()	
winmove()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
winopen()	use auxiliary library or X for windowing	"Replacing winopen() with auxInitWindow()" on page 98 or Chapter 5 and glXIntro reference page
winpop()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
winposition()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
winpush()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
winset()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro and glXMakeCurrent() reference pages
wintitle()	use auxiliary library or X for windowing	"Windowing" on page 96 or Chapter 5 and glXIntro reference page
wmpack()	glColorMask()	"Porting Color, Shading, and Writemask Commands" on page 43
writemask()	glIndexMask()	"Porting Color, Shading, and Writemask Commands" on page 43
writepixels()	glDrawPixels()	
writeRGB()	glDrawPixels()	

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

IRIS GL Call	OpenGL/glu/glX/aux Equivalent	Where Discussed
xfpt*()	not supported	"Porting Picking Calls" on page 81 and "Porting Feedback Calls" on page 87
zbsize()	superseded by selection of an appropriate GLX visual	glXChooseVisual() reference page and "Porting RealityEngine Graphics Features" on page 90
zbuffer()	glEnable(GL_DEPTH_TEST)	
zclear()	glClear(GL_DEPTH_BUFFER_BIT)	"Porting Screen and Buffer Clearing Commands" on page 20
zdraw()	not supported	
zfunction()	glDepthFunc()	
zsource()	not supported	
zwritemask()	glDepthMask()	"Porting Color, Shading, and Writemask Commands" on page 43

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. Each difference is given a simple name, followed by a 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.

attribute lists
: The attributes pushed by IRIS GL **pushattributes()** differ from any of the attribute sets that are pushed by OpenGL **glPushAttrib()**. Because all OpenGL states can be read back, however, it is possible to implement any desired push / pop semantics using the OpenGL API.

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 the *OpenGL Programming Guide* auxiliary library.

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. Also, 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 4-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, IRIS GL's **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 core OpenGL API does not handle concave polygons, but the GLU does provide support for decomposing concave, non-self-intersecting contours into triangles. These triangles can either be draw immediately or returned.

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 included in OpenGL.

curves
: OpenGL does not support IRIS GL curves. Use of NURBS curves is recommended.

defs/binds
: OpenGL 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, however.

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 data base 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. (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 87 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.

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 comprised of triangles can be easily manipulated.

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 mostly handled via X calls. See Chapter 5 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. Also, OpenGL allows separate colors to be specified for the ambient, diffuse, and specular intensities of light sources, as well as 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.

mapw()
: OpenGL utilities don't directly support mapping between object and window coordinates. If you specify the right projection matrix and viewport, you might 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()**.

  Also, material coloring behaves differently: in IRIS GL it was connected with lighting models, while it's part of OpenGL's 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 5, "Mixed-Model Programming." Note in particular that IRIS GL mixed-model routines had, in some cases, names confusingly similar to unrelated OpenGL routines; see "Porting IRIS GL Mixed-Model Programs" in Chapter 5 for details.

move / draw / pmove / pdraw / pclos
: OpenGL supports only Begin / End 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.

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.

NURBS
: OpenGL supports NURBS with a combination of core capability (evaluators) and GLU support. GLU currently supports only Berstein polynomials, not all splines; in the future, GLU may support changing the basis matrix to handle all splines.

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” in Chapter 2.

pick
: The OpenGL utility library includes support for generating a pick matrix.

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.

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 4 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, etc.) 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 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 only true 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 was 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. The IRIS GL **readdisplay()** command may eventually be replaced by an extension to X.

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.

relative move / draw / pmove / pdraw / pclos
: OpenGL does not maintain a current graphics position, and therefore is unable to support relative vertex operations.

reset linestyle
: IRIS GL **resetls()** has not been supported for some time, and is not supported by OpenGL.

RGBA logicop()
: OpenGL does not support 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, though, 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 2-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.

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 the *OpenGL Programming Guide* auxiliary library.

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 lookup tables
: Texture lookup tables aren't yet supported in OpenGL.

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 10 commands for this purpose.

  IRIS GL commands such as **gconfig()** and **drawmode()** are not implemented by OpenGL.

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

Some Important OpenGL Basics

This appendix provides a bit of background information on OpenGL command names, OpenGL defined types, and OpenGL error handling. It is intended mainly as a brief, 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."

OpenGL commands begin with the gl prefix (**glEnable()**, **glTranslatef()**, and so on). OpenGL Utility Library (GLU) commands all begin with the glu prefix (**gluDisk()**, **gluErrorString()**, and so on). The commands that comprise the OpenGL extension to X (GLX) all begin with the glX prefix (**glXChooseVisual()**, **glXCopyContext()**, and so on). The commands that comprise the OpenGL auxiliary library begin with the aux prefix (**auxWireCone()**, **auxSolidTeapot()**, and so on).

OpenGL commands are formed by a root name followed by up to four characters. The first character indicates the number of arguments. The second character, or pair or 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.

The 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 Auxiliary Library

This program uses OpenGL and the auxiliary 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
 */

#include <GL/gl.h>
#include <GL/glu.h>
#include “aux.h”

static int year = 0, day = 0;

void dayAdd (void)
{
    day = (day + 10) % 360;
}

void daySubtract (void)
{
    day = (day - 10) % 360;
}

void yearAdd (void)
{
    year = (year + 5) % 360;
}
```

```
void yearSubtract (void)
{
    year = (year - 5) % 360;
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glColor3f (1.0, 1.0, 1.0);
    glPushMatrix();
/*      draw sun        */
    auxWireSphere(1.0);
/*      draw smaller planet     */
    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);
    auxWireSphere(0.2);
    glPopMatrix();
    glFlush();
}

void myinit (void) {
    glShadeModel (GL_FLAT);
}

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

```
/*  Main Loop
 *  Open window with initial window size, title bar,
 *  RGBA display mode, and handle input events.
 */
int main(int argc, char** argv)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGBA);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (argv[0]);
    myinit ();
    auxKeyFunc (AUX_LEFT, yearSubtract);
    auxKeyFunc (AUX_RIGHT, yearAdd);
    auxKeyFunc (AUX_UP, dayAdd);
    auxKeyFunc (AUX_DOWN, daySubtract);
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
}
```

Appendix E

Example Mixed-Model Program with WorkProc

This appendix contains an example mixed model 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 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();
  }
}
```

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

Here's the IRIS GL version of *iobounce.c*. This is a "pure" IRIS GL program, meaning that 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*

Here's 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*

Here's another example program, *zrgb.c*. This program includes zbuffering. This program won't work on 8-bit IRIS workstations. Again, the IRIS GL version is presented first.

IRIS GL Version of *zrgb.c*

Here's 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*

Here's 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

B

C

D

E

F

G

H

N

O

P

Q

R

S

T

U

V

W

X

Z

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 007-1797-020.

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: 415-965-0964
- 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