Chapter 2

OpenGL and X: Getting Started

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

Background and Terminology

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

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

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

The X Window System on Silicon Graphics Systems

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

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

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

The Silicon Graphics X Server

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

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

The GLX Extension to X

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

Compiling With the GLX Extension

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

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

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

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

X Window System Concepts

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

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

Visuals

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

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

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

When you need visual information,

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

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

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

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

Drawables

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

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

Rendering Contexts

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

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

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

Resources

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

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

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

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

Colormaps

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

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

Libraries, Toolkits, and Tools

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

Widgets and the Xt Library

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

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

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

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

The Xt Library

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

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

For More Information

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

Other Toolkits and Tools

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

Integrating Your OpenGL Program With IRIS IM

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

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

Simple Motif Example Program

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

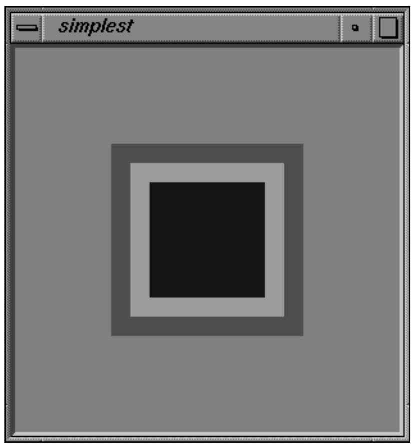

Figure 2-1 Display From simplest.c Example Program

Example 2-1 Simple IRIS IM Program

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

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

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

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

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

/*Process input events*/

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

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

/*Process window resize events*/

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

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

```
/*Process window expose events*/

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

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

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

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

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

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

    XtRealizeWidget(toplevel);

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

    XtAppMainLoop(app);
}
```

Looking at the Example Program

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

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

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

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

Opening the X Display

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

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

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

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

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

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

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

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

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

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

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

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

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

Selecting a Visual

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

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

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

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

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

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

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

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

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

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

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

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

Creating a Rendering Context

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

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

Here's how you use the arguments:

dpy — The display you've already opened.

vis — The visual you've chosen with **glXChooseVisual()**.

sharedList — A context to share display lists with, or NULL to not share display lists.

direct — Lets you specify direct or indirect rendering. For best performance, always request direct rendering. The OpenGL implementation automatically switches to indirect rendering when direct rendering isn't possible (for example, when rendering remotely). See "Direct and Indirect Rendering" on page 79.

Creating the Window

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

Binding the Context to the Window

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

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

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

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

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

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

Mapping the Window

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

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

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

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

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

Summary

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

Table 2-2 Integrating OpenGL and X

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

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

Compiling With OpenGL and Related Libraries

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

Link Lines for Individual Libraries

-lGL OpenGL and GLX routines.

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

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

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

Link Lines for Groups of Libraries

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