OpenGL and X: Examples

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

- "Using Widgets" on page 27 illustrates how to create a window using IRIS IM drawing-area widgets and how to handle input and other events using callbacks.
- "Using Xlib" on page 39 illustrates how to create a colormap and a window for OpenGL drawing. It also provides a brief discussion of event handling with Xlib.

This chapter also briefly discusses fonts: "Using Fonts and Strings" on page 47 looks at a simple example of using fonts with the **glXUseFont()** function.

Note: All integration aspects that aren't dependent on your choice of Xlib or Motif are discussed in "Integrating Your OpenGL Program With IRIS IM" on page 15 in Chapter 2, "OpenGL and X: Getting Started."

Using Widgets

This section explains how to use IRIS IM widgets for creating windows, handling input, and performing other activities the OpenGL part of a program does not deal with. The section discusses the following topics:

- "What are OpenGL Drawing-Area Widgets?"
- "Drawing-Area Widget Setup and Creation"
- "Input Handling With Widgets and Xt"
- "Widget Troubleshooting"

What are OpenGL Drawing-Area Widgets?

Using an OpenGL drawing-area widget facilitates rendering OpenGL into an X window. The widget

- provides an environment for OpenGL rendering, including a visual and a colormap
- provides a set of callback routines for redrawing, resizing, input, and initialization (see "Using Drawing-Area Widget Callbacks" on page 32)

OpenGL provides two drawing-area widgets: GLwMDrawingArea—note the M in the name—for use with IRIS IM (or with OSF/Motif), and GLwDrawingArea for use with any other widget sets. Both drawing-area widgets provide two convenience functions:

- **GLwMDrawingAreaMakeCurrent()** and **GLwDrawingAreaMakeCurrent()**
- **GLwMDrawingAreaSwapBuffers()** and **GLwDrawingAreaSwapBuffers()**

The functions allow you to supply a widget instead of the display and window required by the corresponding GLX functions **glXMakeCurrent()** and **glXSwapBuffers()**.

Because the two widgets are nearly identical, and because IRIS IM is available on all Silicon Graphics systems, this chapter uses only the IRIS IM version, even though most of the information also applies to the general version. Here are some of the distinguishing characteristics of GLwMDrawingArea:

- GLwMDrawingArea understands IRIS IM keyboard traversal (moving around widgets with keyboard keys rather than a mouse), although keyboard traversal is turned off by default.
- GLwMDrawingArea is a subclass of the IRIS IM XmPrimitive widget, not a subclass of the Xt Core widget. It therefore has various defaults such as background and foreground colors. GLwMDrawingArea is **not** derived from the standard Motif drawing-area widget class. (See O'Reilly Volume One or the reference pages for Core and for XmPrimitive for more information.)

  Note that the default background colors provided by the widget are used during X rendering, not during OpenGL rendering, so it's not advisable to rely on default background rendering from the widget. Even when the background colors are not used directly, **XtGetValues()** can be used to query them to allow the graphics to blend in better with the program.
- GLwMDrawingArea has an IRIS IM style creation function, **GLwCreateMDrawingArea()**; you can also create the widget directly through Xt.

For information specific to GLwDrawingArea, see the reference page.

Drawing-Area Widget Setup and Creation

Most of the steps for writing a program that uses a GLwMDrawingArea widget are already discussed in "Integrating Your OpenGL Program With IRIS IM" on page 15. This section explains how to initialize IRIS IM and how to create the drawing-area widget, using code fragments from the *motif/simplest.c* example program (Example 2-1 on page 16). You learn about

- "Setting Up Fallback Resources"
- "Creating the Widgets"
- "Choosing the Visual for the Drawing-Area Widget"
- "Creating Multiple Widgets With Identical Characteristics"
- "Using Drawing-Area Widget Callbacks"

Setting Up Fallback Resources

This section briefly explains how to work with resources in the context of an OpenGL program. In Xt, resources provide widget properties, allowing you to customize how your widgets will look. Note that the term "resource" used here refers to window properties stored by a resource manager in a resource database, not to the data structures for windows, pixmaps, and context discussed earlier.

Fallback resources inside a program are used when a widget is created and the application can't open the class resource file when it calls **XtOpenApplication()** to open the connection to the X server. (In the code fragment below, the first two resources are specific to Silicon Graphics and give the application a Silicon Graphics look and feel.)

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

Note: Applications should ship with resource files installed in a resource directory (in */usr/lib/X11/app-defaults*). If you do install such a file automatically with your application, there is no need to duplicate the resources in your program.

Creating the Widgets

Widgets always exist in a hierarchy, with each widget contributing to what's visible on screen. There's always a top-level widget and almost always a container widget (for example, form or frame). In addition, you may decide to add buttons or scroll bars, which are also part of the IRIS IM widget set. Creating your drawing surface therefore consists of two steps:

1. Create parent widgets, namely the top-level widget and a container widget. *motif/simplest.c,* Example 2-1 on page 16, uses a Form container widget and a Frame widget to draw the 3D box:

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

   For more information, see the reference pages for XmForm and XmFrame.

2. Create the GLwMDrawingArea widget itself in either of two ways:

   - Call **GLwCreateMDrawingArea()**. You can specify each attribute as an individual resource or pass in an XVisualInfo pointer obtained with **glXChooseVisual()**. This is discussed in more detail in the next section, "Choosing the Visual for the Drawing-Area Widget."

     ```
     n = 0
     XSetArg(args[n] GLwNvisualinfo, (XtArgVal)visinfo);
     n++;
     glw = GLwCreateMDrawingArea(frame, "glwidget", args, n);
     ```

   - As an alternative, call **XtVaCreateManagedWidget()** and pass it a pointer to the visual you've chosen. In that case, use glwMDrawingAreaWidgetClass as the parent and GLwNvisualInfo to specify the pointer. Here's an example from *motif/simplest.c:*

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

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

Note that unlike most other Motif widgets, the OpenGL widget explicitly sets the visual. Once a visual is set and the widget is realized, the visual can no longer be changed.

Choosing the Visual for the Drawing-Area Widget

There are three ways of configuring the GLwMDrawingArea widget when calling the widget creation function, all done through resources:

- Pass in separate resources for each attribute (for example GLwNrgba, GLwNdoublebuffer).
- Pass in an attribute list of the type used by **glXChooseVisual(),** using the GLwNattribList resource.
- Select the visual yourself, using **glXChooseVisual(),** and pass in the returned XvisualInfo as the GLwNvisualInfo resource.

If you wish to provide error handling, call **glXChooseVisual()**, as all the example programs do (although for the sake of brevity, none of the examples actually provides error handling). If you provide the resources and let the widget choose the visual, the widget just prints an error message and quits. Note that a certain visual may be supported on one system but not on another, so appropriate error handling is critical to a robust program.

The advantage of using a list of resources is that you can override them with the *app-defaults* file.

Creating Multiple Widgets With Identical Characteristics

Most applications have one context per widget, though sharing is possible. If you want to use multiple widgets with the same configuration, you must use the same visual for each widget. Windows with different visuals can't share contexts. To share contexts:

1. Extract the GLwNvisualInfo resource from the first widget you create.
2. Use that visual in the creation of subsequent widgets.

Using Drawing-Area Widget Callbacks

The GLwMDrawingArea widget provides callbacks for redrawing, resizing, input, and initialization, as well as the standard XmNdestroyCallback provided by all widgets.

Each callback must first be defined and then added to the widget. In some cases, this is quite simple, as, for example, the resize callback from *motif/simplest.c*:

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

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

Other cases are slightly more complex, such as the input callback from *motif/simplest.c*, which exits when the user presses the **<Esc>** key:

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

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

To add callbacks to a widget, use **XtAddCallback()**; for example:

```
XtAddCallback(glxwidget, GLwNexposeCallback, expose, NULL);
XtAddCallback(glxwidget, GLwNresizeCallback, resize, NULL);
XtAddCallback(glxwidget, GLwNinputCallback, input, NULL);
```

Each callback must ensure that the thread is made current with the correct context to the window associated with the widget generating the callback. You can do this by calling either **GLwMDrawingAreaMakeCurrent()** or **glXMakeCurrent()**.

If you're using only one GLwMDrawingArea, you can call a routine to make the widget "current" just once, after initializing the widget. However, if you're using more than one GLwMDrawingArea or rendering context, you need to make the correct context and the window current for each callback (see "Binding the Context to the Window" on page 22).

The following callbacks are available:

- **GLwNginitCallback**. Specifies the callbacks to be called when the widget is first realized. You can use this callback to perform OpenGL initialization, such as creating a context, since no OpenGL operations can be done before the widget is realized. Callback reason is GLwCR_GINIT.

  Use of this callback is not necessary. Anything done in this callback can also be done after the widget hierarchy has been realized. You can use the callback to keep all the OpenGL code together, keeping the initialization in the same file as the widget creation rather than with widget realization.

  If you create a GLwDrawingArea widget as a child of an already realized widget, it's not possible to add the ginit callback before the widget is realized because the widget is immediately realized at creation. In that case, you should initialize immediately after creating the widget.

- **GLwNexposeCallback**. Specifies the callbacks to be called when the widget receives an Expose event. The callback reason is GLwCR_EXPOSE. The callback structure also includes information about the Expose event. Usually the application should redraw the scene whenever this callback is called.

  Note: An application should not perform any OpenGL drawing until it receives an expose callback, although it may set the OpenGL state; for example, it may create display lists and so on.

- **GLwNinputCallback**. Specifies the callbacks to be called when the widget receives a keyboard or mouse event. The callback structure includes information about the input event. Callback reason is GLwCR_INPUT.

  The input callback is a programming convenience; it provides a convenient way to catch all input events. You can often create a more modular program, however, by providing specific actions and translations in the application rather than using a single catchall callback. See "Input Handling With Widgets and Xt" on page 34 for more information.

- **GLwNresizeCallback**. Specifies the callbacks to be called when the GLwDrawingArea is resized. The callback reason is GLwCR_RESIZE. Normally, programs resize the OpenGL viewport and possibly reload the OpenGL projection matrix (see the *OpenGL Programming Guide*). An expose callback follows. Avoid performing rendering inside the resize callback.

Input Handling With Widgets and Xt

This section explains how to perform input handling with widgets and Xt. It covers:

- "Background Information"
- "Using the Input Callback"
- "Using Actions and Translations"

Background Information

Motif programs are callback driven. They differ in that respect from IRIS GL programs, which implement their own event loops to process events. To handle input with a widget, you can either use the input callback built into the widget or use actions and translations (Xt-provided mechanisms that map keyboard input into user-provided routines). Both approaches have advantages:

- Input callbacks are usually simpler to write, and they're more unified; all input is handled by a single routine that can maintain a private state (see "Using the Input Callback").
- The actions-and-translations method is more modular, because translations have one function for each action. Also, with translations the system does the keyboard parsing so your program doesn't have to do it. Finally, translations allow the user to customize the application's key bindings. See "Using Actions and Translations" on page 36.

Note: To allow smooth porting to other systems, as well as for easier integration of X and OpenGL, always separate event handling from the rest of your program.

Using the Input Callback

By default, the input callback is called with every key press and release, with every mouse button press and release, and whenever the mouse is moved while a mouse button is pressed. You can change this by providing a different translation table, although the default setting should be suitable for most applications.

For example, to have the input callback called on all pointer motions, not just on mouse button presses, add the following to the *app-defaults* file:

```
appname*widgetname.translations : \
    <KeyDown>:      glwInput() \n\
    <KeyUp>:        glwInput() \n\
    <BtnDown>:      glwInput() \n\
    <BtnUp>:        glwInput() \n\
    <BtnMotion>:    glwInput() \n\
    <PtrMoved>:     glwInput()
```

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

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

Example 3-1 Motif Program That Handles Mouse Events

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

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

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

Using Actions and Translations

Actions and translations provide a mechanism for binding a key or mouse event to a function call. For example, you can set things up so that

- when you press the <Esc> key, the exit routine **quit()** is called
- when you press the left mouse button, rotation occurs
- when you press f, the program zooms in

The translations need to be combined with an action task that maps string names like quit() to real function pointers. Below is an example of a translation table:

```
program*glwidget*translations:      #override \n
    <Btn1Down>:         start_rotate()   \n\
    <Btn1Up>:           stop_rotate()    \n\
    <Btn1Motion>:       rotate()         \n\
    <Key>f:             zoom_in()        \n\
    <Key>b:             zoom_out()       \n\
    <KeyUp>osfCancel:   quit()
```

When you press the left mouse button, the **start_rotate()** action is called; when it is released, the **stop_rotate()** action is called.

The last entry is a little cryptic. It actually says that when the user presses the <Esc> key, **quit()** is called. However, OSF has implemented virtual bindings, which allow the same programs to work on computers with different keyboards that may be missing various keys. If a key has a virtual binding, the virtual binding name must be specified in the translation. Thus, the example above specifies osfCancel rather than <Esc>. To use the above translation in a program that is not based on IRIS IM or OSF/Motif, replace <KeyUp>osfCancel with <KeyUp><Esc>.

The translation is only half of what it takes to set up this binding. Although the translation table above contains what look like function names, they're really action names. Your program must also create an action table to bind the action names to actual functions in the program.

For more information on actions and translations, see O'Reilly, *X Toolkit Intrinsics Programming Manual* (Volume 4), most notably Chapter 4, "An Example Application," and Chapter 8, "Events, Translations, and Accelerators." You can view this manual online using IRIS InSight.

Creating Colormaps

By default, a widget creates a colormap automatically. For many programs, this is sufficient. However, it is occasionally necessary to create a colormap explicitly, especially when using color index mode. See "Creating a Colormap and a Window" on page 42 and "Using Colormaps" on page 69 for more information.

Widget Troubleshooting

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

Note: Additional debugging information is provided in "Tips for Debugging Graphics Programs" on page 205.

Keyboard Input Disappears

A common problem in IRIS IM programs is that keyboard input disappears. This is caused by how IRIS IM handles keyboard focus. When a widget hierarchy has keyboard focus, only one component of the hierarchy receives the keyboard events. The keyboard input might be going to the wrong widget.

There are two solutions to this:

- The easiest solution is to set the resource

  ```
  keyboardFocusPolicy: POINTER
  ```

  for the application. This overrides the default traversal method (explicit traversal) where you can select widgets with keyboard keys rather than the mouse so that input focus follows the pointer only. The disadvantages of this method are that it eliminates explicit traversal for users who prefer it and it forces a nondefault model.

- A better solution is to set the resource

  ***widget*.traversalOn: TRUE*

  where *widget* is the name of the widget, and to call

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

  whenever mouse button 1 is pressed in the widget. Turning process traversal on causes the window to respond to traversal (it normally does not), and calling **XmProcessTraversal()** actually traverses into the widget when appropriate.

Inheritance Issues

In Xt, shell widgets, which include top-level windows, popup windows, and menus,

- inherit their colormap and depth from their parent widget
- inherit their visual from the parent window

If the visual does not match the colormap and depth, this leads to a BadMatch X protocol error.

In a typical IRIS IM program, everything runs in the default visual, and the inheritance from two different places does not cause problems. However, when a program uses both OpenGL and IRIS IM, it requires multiple visuals, and you have to be careful. Whenever you create a shell widget as a child of a widget in a non-default visual, specify pixel depth, colormap, and visual for that widget explicitly. This happens with menus or popup windows that are children of OpenGL widgets. See “Using Popup Menus With the GLwMDrawingArea Widget” on page 64.

If you do get a BadMatch error, follow these steps to determine its cause:

1. Run the application under a C debugger, such as dbx or cvd (the Case Vision debugger) with the **-sync** flag.

   The **-sync** flag tells Xt to call **XSynchronize()**, forcing all calls to be made synchronously. If your program is not Xt-based, or if you aren’t using standard argument parsing, call **XSynchronize(display, TRUE)** directly inside your program.

2. Using the debugger, set a breakpoint in **exit()** and run the program.

   When the program fails, you have a stack trace you can use to determine the problem.

Note: If you don’t use the **-sync** option, the stack dump on failure is meaningless: X batches multiple requests and the error is delayed.

Using Xlib

This section explains how to use Xlib for creating windows, handling input, and performing other activities the OpenGL part of a program does not deal with. Since the complete example program in Chapter 2, “OpenGL and X: Getting Started” used widgets, this section starts with a complete annotated example program for Xlib, so you have both available as needed. After that, you learn about

- Creating a Colormap and a Window
- Xlib Event Handling

Simple Xlib Example Program

Example 3-2 lists the complete *Xlib/simplest.c* example program.

Example 3-2 Simple Xlib Example Program

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

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

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

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

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

    do {
        char buf[31];
        KeySym keysym;

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

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

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

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

    /* get an appropriate visual */
    vi = glXChooseVisual(dpy, DefaultScreen(dpy), attributeList);
    if (!vi) error(argv[0], "no suitable visual");

    /* create a GLX context */
    cx = glXCreateContext(dpy, vi, 0, GL_TRUE);

    /* create a colormap */
    swa.colormap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
                                   vi->visual, AllocNone);
    /* create a window */
    swa.border_pixel = 0;
    swa.event_mask = ExposureMask | StructureNotifyMask | KeyPressMask;
    win = XCreateWindow(dpy, RootWindow(dpy, vi->screen), 0, 0, 300,
                        300, 0, vi->depth, InputOutput, vi->visual,
                        CWBorderPixel|CWColormap|CWEventMask, &swa);
```

```
    XStoreName(dpy, win, "simplest");
    XMapWindow(dpy, win);

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

    while (1) process_input(dpy);
}
```

Creating a Colormap and a Window

A colormap determines the mapping of pixel values in the framebuffer to color values on the screen. Colormaps are created with respect to a specific visual.

When you create a window, you must supply a colormap for it. The visual associated with a colormap must match the visual of the window using the colormap. Most X programs use the default colormap because most X programs use the default visual. The easiest way to obtain the colormap for a particular visual is to call **XCreateColormap()**:

```
Colormap XCreateColormap (Display *display, Window w, Visual *visual,
                          int alloc)
```

Here's how Example 3-2 calls **XCreateColormap()**:

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

The parameters specify the display, window, and visual, and the number of colormap entries to allocate. The *alloc* parameter can have the special value AllocAll or AllocNone. While it is easy to simply call **XCreateColormap()**, you are encouraged to share colormaps. See Example 4-2 on page 71 for details on how to do this.

Note that you can't use AllocAll if the colormap corresponds to a visual that has transparent pixels, because the colormap cell that corresponds to the transparent pixel cannot be allocated with AllocAll. For more information about colormaps, see "Using Colormaps" on page 69. For information on overlays, which use a visual with a transparent pixel, see "Using Overlays" on page 58.

You can then create a window using **XCreateWindow()**. Before calling **XCreateWindow()**, set the attributes you want in the *attributes* variable. When you make the call, indicate *valuemask* by OR-ing together symbolic constants that specify the attributes you've set. Here's how Example 3-2 does it:

```
swa.background_pixmap = None;
swa.border_pixel = 0;
swa.event_mask = ExposureMask | StructureNotifyMask | KeyPressMask;
win = XCreateWindow(
             dpy,                                /*display*/
             RootWindow(dpy, vi->screen),        /*parent*/
             0,                                  /*x coordinate*/
             0,                                  /*y coordinate*/
             300,                                /*width*/
             300,                                /*height*/
             0,                                  /*border width*/
             vi->depth,                          /*depth*/
             InputOutput,                        /*class*/
             vi->visual,                         /*visual*/
             CWBackPixmap|CWBorderPixel|CWColormap|CWEventMask,
                                                 /*valuemask*/
             &swa                                /*attributes*/
);
```

Most of the parameters are self-explanatory. Here are three that are not:

- *class* indicates whether the window is InputOnly or InputOutput.

  Note: InputOnly windows cannot be used with GLX contexts.

- *valuemask* specifies which window attributes are provided by the call.
- *attributes* specifies the settings for the window attributes. The XSetWindowAttributes structure contains a field for each of the allowable attributes.

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

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

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

XSetWMProperties() provides a convenient interface for setting a variety of important window properties at once. It merely calls a series of other property-setting functions, passing along the values you pass in. For more information, see the reference page.

Note that two useful properties are the window name and the icon name. The example program calls **XStoreName()** instead to set the window and icon names.

Installing the Colormap

Applications should rely on the window manager to install the colormaps instead of calling **XInstallColormap()** directly. The window manager automatically installs the appropriate colormaps for a window, whenever that window gets keyboard focus.

By default, the window manager looks at the top-level window of a window hierarchy and installs that colormap when the window gets keyboard focus. For a typical X-based application, this is sufficient, but an application based on OpenGL typically uses multiple colormaps: the top-level window uses the default X colormap, and the Open GL window uses a colormap suitable for OpenGL.

To address this multiple colormap issue, call the function **XSetWMColormapWindows()** passing the display, the top-level window, a list of windows whose colormaps should be installed, and the number of windows in the list.

The list of windows should include one window for each colormap, including the top-level window's colormap (normally represented by the top-level window). For a typical OpenGL program that does not use overlays, the list contains two windows: the OpenGL window and the top-level window. The top-level window should normally be last in the list. Xt programs may use **XtSetWMColormapWindows()** instead of **XSetWMColormapWindows()**, which uses widgets instead of windows.

Note: The program must call **XSetWMColormapWindows()** even if it is using a TrueColor visual. Some hardware simulates TrueColor through the use of a colormap. Even though the application doesn't interact with the colormap directly, it is still there. If you don't call **XSetWMColormapWindows(),** your program may run correctly only some of the time, and only on some systems.

Use the *xprop* program to determine whether **XSetWMColormapWindows()** was called. Click the window and look for the WM_COLORMAP_WINDOWS property. This should be a list of the windows. The last one should be the top-level window. Use *xwininfo,* providing the ID of the window as an argument, to determine what colormap the specified window is using, and whether that colormap is installed.

Xlib Event Handling

This section discusses different kinds of user input and explains how you can use Xlib to perform them. OpenGL programs running under the X Window System are responsible for responding to events sent by the X server. Examples of X events are Expose, ButtonPress, ConfigureNotify, and so on.

Note: In addition to mouse devices, Silicon Graphics systems support various other input devices (for example, spaceballs). You can integrate them with your OpenGL program using the X input extension. For more information, see the *X Input Extension Library Specification* available online through IRIS Insight.

Handling Mouse Events

To handle mouse events, your program first has to request them, then use them in the main (event handling) loop. Here's an example code fragment from *Xlib/mouse.c*, an Xlib program that uses mouse motion events. Example 3-3 shows how the mouse processing, along with the other event processing, is defined.

Example 3-3 Event Handling With Xlib

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

    do {
        char buf[31];
        KeySym keysym;

        XNextEvent(dpy, &event);
        switch(event.type) {
        case Expose:
            redraw = 1;
            break;
        case ConfigureNotify:
            glViewport(0, 0, event.xconfigure.width,
                       event.xconfigure.height);
            redraw = 1;
            break;
```

```
        case KeyPress:
            (void) XLookupString(&event.xkey, buf, sizeof(buf),
                  &keysym, NULL);
            switch (keysym) {
            case XK_Escape:
                exit(EXIT_SUCCESS);
            default:
                break;
            }
        case ButtonPress:
            if (event.xbutton.button == Button2) {
                mstate |= 2;
                mx = event.xbutton.x;
                my = event.xbutton.y;
            } else if (event.xbutton.button == Button1) {
                mstate |= 1;
                mx = event.xbutton.x;
                my = event.xbutton.y;
            }
            break;
        case ButtonRelease:
            if (event.xbutton.button == Button2)
                mstate &= ~2;
            else if (event.xbutton.button == Button1)
                mstate &= ~1;
            break;
        case MotionNotify:
            if (mstate) {
                omx = mx;
                omy = my;
                mx = event.xbutton.x;
                my = event.xbutton.y;
                update_view(mstate, omx,mx,omy,my);
                redraw = 1;
            }
            break;
        default:
            break;
        }
    } while (XPending(dpy));
    return redraw;
}
```

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

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

Exposing a Window

When a user selects a window that has been completely or partly covered, the X server generates one or more Expose events. It is difficult to determine exactly what was drawn in the now-exposed region and redraw only that portion of the window. Instead, OpenGL programs usually just redraw the entire window. (Note that backing store is not supported on Silicon Graphics systems.)

If redrawing is not an acceptable solution, the OpenGL program can do all your rendering into a GLXPixmap instead of directly to the window; then, any time the program needs to redraw the window, you can simply copy the GLXPixmap's contents into the window using **XCopyArea()**. For more information, see "Using Pixmaps" on page 77.

Caution: Rendering to a GLXPixmap is much slower than rendering to a window. For example, on a RealityEngine, rendering to a pixmap is perhaps 5% the speed of rendering to a window.

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

Using Fonts and Strings

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

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

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

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

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

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

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

Note: You can also use the glc library, which sits atop of OpenGL, for fonts and strings. The library is not specific to GLX and lets you do more than **glXUseXFont().**

Example 3-4 Font and Text Handling

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

GLuint base;

void makeRasterFont(Display *dpy)
{
    XFontStruct *fontInfo;
    Font id;
    unsigned int first, last;
    fontInfo = XLoadQueryFont(dpy,
        "-adobe-times-medium-r-normal--17-120-100-100-p-88-iso8859-1");
```

```
if (fontInfo == NULL) {
        printf ("no font found\n");
        exit (0);
    }

    id = fontInfo->fid;
    first = fontInfo->min_char_or_byte2;
    last = fontInfo->max_char_or_byte2;

    base = glGenLists(last+1);
    if (base == 0) {
        printf ("out of display lists\n");
    exit (0);
    }
    glXUseXFont(id, first, last-first+1, base+first);
}

void printString(char *s)
{
    glListBase(base);
    glCallLists(strlen(s), GL_UNSIGNED_BYTE, (unsigned char *)s);
}

void display(void)
{
    GLfloat white[3] = { 1.0, 1.0, 1.0 };
    long i, j;
    char teststring[33];

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3fv(white);
    for (i = 32; i < 127; i += 32) {
        glRasterPos2i(20, 200 - 18*i/32);
        for (j = 0; j < 32; j++)
            teststring[j] = i+j;
        teststring[32] = 0;
        printString(teststring);
    }
    glRasterPos2i(20, 100);
    printString("The quick brown fox jumps");
    glRasterPos2i(20, 82);
    printString("over a lazy dog.");
    glFlush ();
}
```