*Chapter 4*

# OpenGL and X: Advanced Topics

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



## Using Animations

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

This section discusses double-buffered animation inside an X Window System environment, providing example code as appropriate. You learn about

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

Xt provides two mechanisms that are suited for continuous animation:

- "Controlling an Animation With Workprocs" on page 53 results in the fastest animation possible. If you use workprocs, the program swaps buffers as fast as possible; which is useful if rendering speed is variable enough that constant speed animation is not possible. Workproc animations also give other parts of the application priority. The controls don't become less responsive just because the animation is being done. The cost of this is that the animation slows down or may stop when the user brings up a menu or uses other controls.
- "Controlling an Animation With Timeouts" on page 56 results in a constant speed animation. Animations that use timeouts compete on even footing with other Xt events; the animation won't stop because the user interacts with other components of the animation.

**Note:** Controlling animations with workprocs and timeouts applies only to Xt-based programs.

## Swapping Buffers

A double-buffered animation displays one buffer while drawing into another (undisplayed) buffer, then swaps the displayed buffer with the other. In OpenGL, the displayed buffer is called the front buffer, and the undisplayed buffer is called the back buffer. This sort of action is common in OpenGL programs; however, swapping buffers is a window-related function, not a rendering function, so you can't do it directly with OpenGL.

To swap buffers, use **glXSwapBuffers()** or (when using the widget) the convenience function **GLwDrawingAreaSwapBuffers()**. The **glXSwapBuffers()** function takes a display and a window as input—pixmaps don't support buffer swapping—and swaps the front and back buffers in the drawable. All renderers bound to the window in question continue to have the correct idea of which is the front buffer and which the back buffer. Note that once you call **glXSwapBuffers()**, any further drawing to the given window is suspended until after the buffers have been swapped.

Silicon Graphics systems support hardware double buffering; this means buffer swap is instantaneous during the vertical retrace of the monitor. As a result, there are no tearing artifacts; that is, you don't simultaneously see part of one buffer and part of the next.

**Caution:** If the window's visual allows only one color buffer, or if the GLX drawable is a pixmap, **glXSwapBuffers()** has no effect (and generates no error).

There’s no need to worry about which buffer the X server draws into if you’re using X drawing functions as well as OpenGL; the X server draws only to the current front buffer, and prevents any program from swapping buffers while such drawing is going on. Using the X double buffering extension (DBE), it is possible to render X into the back buffer. DBE is not supported in releases preceding IRIX 6.2.

Note that users like uniform frame rates such as 60 Hz, 30 Hz, or 20 Hz. Animation may otherwise look jerky. A slower consistent rate is therefore preferable to a faster but inconsistent rate. For additional information about optimizing frame rates, see “Optimizing Frame Rate Performance” on page 215. See “The Swap Control Extension” on page 168 to learn how to set a minimum period of buffer swaps.

## Controlling an Animation With Workprocs

A workproc (work procedure) is a procedure that Xt calls when the application is idle. The application registers workprocs with Xt and unregisters them when it’s time to stop calling them.

Note that workprocs do not provide constant speed animation but animate as fast as the application can.

### General Workproc Information

Workprocs can be used to carry out a variety of useful tasks: animation, setting up widgets in the background (to improve application startup time), keeping a file up to date, and so on.

It’s important that a workproc not take very long to execute. While a workproc is running, nothing else can run, and the application may appear sluggish or may even appear to hang.

Workprocs return Booleans. To set up a function as a workproc, first prototype the function, then pass its name to **XtAppAddWorkProc()**. Xt then calls the function whenever there’s idle time while Xt is waiting for an event. If the function returns True, it’s removed from the list of workprocs; if it returns False, it’s kept on the list and called again when there’s idle time.

To explicitly remove a workproc, call **XtRemoveWorkProc()**. Here are the prototypes for the add and remove functions:

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

void XtRemoveWorkProc(XtWorkProcId id)
```

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

### Workproc Example

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

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

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

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

```
static void
map_change(Widget w, XtPointer clientData, XEvent *event, Boolean
                                                             *cont) {
    switch (event->type) {
```

```
    case MapNotify:
    /* resume animation if we become mapped in the animated state */
        if (state.animate_wpid != 0)
             state.animate_wpid = XtAppAddWorkProc(state.appctx,
                                        redraw_proc, &state.glxwidget);
        break;
    case UnmapNotify:
    /* don't animate if we aren't mapped */
        if (state.animate_wpid) XtRemoveWorkProc(state.animate_wpid);
        break;
    }
}
```

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

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

The **redraw_proc()** function, in turn, calls **draw_scene()**, which swaps the buffers. Note that this program doesn't use **glXSwapBuffers()**, but instead the convenience function **GLwDrawingAreaSwapBuffers()**.

```
static void
draw_scene(Widget w) {
    static float rot = 0.;

    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(.1, .1, .8);
    glPushMatrix();
    if ((rot += 5.) > 360.) rot -= 360.;
    glRotatef(rot,0.,1.,0.);
    cube();
    glScalef(0.3,0.3,0.3);
    glColor3f(.8, .8, .1);
    cube();
    glPopMatrix();
    GLwDrawingAreaSwapBuffers(w);
}
```

**Note:** If an animation is running and the user selects a menu command, the event handling for the command and the animation may end up in a race condition.

### Controlling an Animation With Timeouts

The program that performs an animation using timeouts is actually quite similar to the one using workprocs. The main difference is that the timeout interval has to be defined and functions that relied on the workproc now have to be defined to rely on the timeout. Note especially that **redraw_proc()** has to register a new timeout each time it's called.

You may find it most helpful to compare the full programs using *xdiff* or a similar tool. This section briefly points out the main differences between two example programs.

- The redraw procedure is defined to have an additional argument, an interval ID.

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

  **time_animate**:
  ```
  static Boolean redraw_proc(XtPointer clientData,
                             XtIntervalId *id);
  ```

- In *time_animate,* a timeout has to be defined; the example chooses 10 ms:

  ```
  #define TIMEOUT 10 /*timeout in milliseconds*/
  ```

- In the state structure, which defines the global UI variables, the interval ID instead of the workproc ID is included.

  **work_animate**:

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

  **time_animate**:

  ```
  static struct {        /* global UI variables; keep them together */
      XtAppContext appctx;
      Widget glxwidget;
      Boolean direct;
      XtIntervalId animate_toid;
  } state;
  ```

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

  **work_animate:**

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

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

  **time_animate**

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

    switch (entry) {
    case 0:
        if (state.animate_toid) {
            XtRemoveTimeOut(state.animate_toid);
            state.animate_toid = 0;
        } else {
            /* register timeout */
            state.animate_toid = XtAppAddTimeOut(state.appctx,
                            TIMEOUT, redraw_proc, &state.glxwidget);
        }
        break;
```

```
        case 1:
            exit(EXIT_SUCCESS);
            break;
        default:
            break;
        }
    }
```

- The **redraw_proc()** function has to register a new timeout each time it's called. Note that this differs from the workproc approach, where the application automatically continues animating as long as the system isn't doing something else.

```
static void
redraw_proc(XtPointer clientData, XtIntervalId *id) {
    Widget *w = (Widget *)clientData;
    draw_scene(*w);
    /* register a new timeout */
    state.animate_toid = XtAppAddTimeOut(state.appctx, TIMEOUT,
                                    redraw_proc, &state.glxwidget);
}
```

## Using Overlays

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

- "Introduction to Overlays"
- "How to Create Overlays"
- "Rubber Banding"

### Introduction to Overlays

An overlay plane is a set of bitplanes displayed preferentially to the normal planes. Non-transparent pixels in the overlay plane are displayed in preference to the underlying pixels in the normal planes. Windows in the overlay planes do not damage windows in the normal plane.

If you have something in the main window that's fairly expensive to draw into and want to have something else on top, such as an annotation, you can use a transparent overlay plane to avoid redrawing the more expensive main window. Overlays are well-suited for popup menus, dialog boxes, and "rubber-band" image resizing rectangles. You can also use overlay planes for text annotations floating "over" an image and for certain transparency effects.

**Note:** Transparency discussed here is distinct from alpha buffer blending transparency effects. See the section "Blending" in Chapter 7, "Blending, Anti-Aliasing, and Fog," in the *OpenGL Programming Guide.*

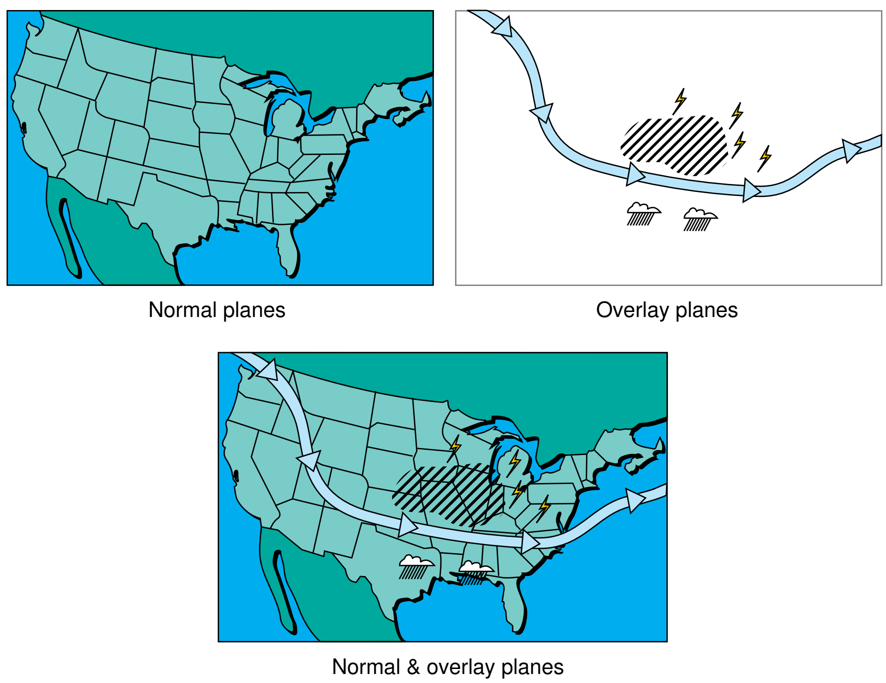


**Figure 4-1** Overlay Plane Used for Transient Information

A special value in the overlay planes indicates transparency. On Silicon Graphics systems, it's always the value zero. Any pixel with the value zero in the overlay plane is not painted, allowing the color of the corresponding pixel in the normal planes to show.

The concepts discussed in this section apply more generally to any number of framebuffer layers, for example, underlay planes (which are covered up by anything in equivalent regions of higher-level planes).

You can use overlays in two ways:

- To draw additional graphics in the overlay plane on top of your normal plane OpenGL widget, create a separate GLwMDrawingArea widget in the overlay plane and set the GLX_LEVEL resource to 1. Position the overlay widget on top of the normal plane widget.

  Note that since the GLwMDrawingArea widget is not a manager widget, it is necessary to create both the normal and overlay widgets as children of some manager widget—for example, a form—and have that widget position the two on top of each other. Once the windows are realized, you must call **XRaiseWindow()** to guarantee that the overlay widget is on top of the normal widget. Code fragments in "How to Create Overlays" on page 60 illustrate this. The whole program is included as *overlay.c* in the source tree.
- To create menus, look at examples in */usr/src/X11/motif/overlay_demos*. They are present if you have the *motif_dev.sw.demo* subsystem installed. Placing the menus in the overlay plane avoids the need for expensive redrawing of the OpenGL window underneath them. While the demos do not deal specifically with OpenGL, they do show how to place menus in the overlay plane.

### A Note for IRIS GL Users

IRIS GL supports the concept of popup planes, which are one level higher than the default overlay plane. Drawing in the popup planes in IRIS GL doesn't necessarily require a window, but you can't count on avoiding damage to anything non-transient drawn in those planes (for example, objects drawn by other applications).

When working with OpenGL and the X Window System, the situation is different: You have to create a separate window for any overlay rendering. Currently, no OpenGL implementation on a Silicon Graphics system supports a level greater than one.

## How to Create Overlays

This section explains how to create overlay planes, using an example program based on Motif. If you create the window using Xlib, the same process is valid (and a parallel example program is available in the example program directory).

The example program from which the code fragments are taken, *motif/overlay.c,* uses the visual info extension to find a visual with a transparent pixel. See "The Visual Info Extension" on page 165 for more information.

**Note:** This example does not work if the visual info extension is not available (see "How to Check for OpenGL Extension Availability" on page 82). The visual info extension is available only in IRIX 6.2. In IRIX 5.3 and earlier releases, you must look at the TRANSPARENT_OVERLAYS property on the root window to get the information.

To create the overlay, follow these steps:

1. Define attribute lists for the two widgets (the window and the overlay). For the overlay, specify GLX_LEVEL as 1 and GLX_TRANSPARENT_TYPE_EXT as GLX_TRANSPARENT_RGB_EXT if the visual info extension is available.

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

2. Create a frame and form, then create the window widget, attaching it to the form on all four sides. Add expose, resize, and input callbacks.

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

/* attach to form on all 4 sides */
n = 0;
XtSetArg(args[n], XtNx, 0); n++;
XtSetArg(args[n], XtNy, 0); n++;
XtSetArg(args[n], XmNtopAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNrightAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(args[n], GLwNvisualInfo, visinfo); n++;
state.w = XtCreateManagedWidget("glxwidget",
    glwMDrawingAreaWidgetClass, form, args, n);
XtAddCallback(state.w, GLwNexposeCallback, expose, NULL);
XtAddCallback(state.w, GLwNresizeCallback, resize, &state);
XtAddCallback(state.w, GLwNinputCallback, input, NULL);
state.cx = glXCreateContext(dpy, visinfo, 0, GL_TRUE);
```

3. Create the overlay widget, using the overlay visual attributes specified in Step 1 and attaching it to the same form as the window. This assures that when the window is moved or resized, the overlay is as well.

```
if (!(visinfo = glXChooseVisual(dpy, DefaultScreen(dpy),
    ov_attribs)))
    XtAppError(appctx, "no suitable overlay visual");
XtSetArg(args[n-1], GLwNvisualInfo, visinfo);
ov_state.w = XtCreateManagedWidget("overlay",
    glwMDrawingAreaWidgetClass, form, args, n);
```

4. Add callbacks to the overlay.

```
XtAddCallback(ov_state.w, GLwNexposeCallback, ov_expose, NULL);
XtAddCallback(ov_state.w, GLwNresizeCallback, resize, &ov_state);
XtAddCallback(ov_state.w, GLwNinputCallback, input, NULL);
ov_state.cx = glXCreateContext(dpy, visinfo, 0, GL_TRUE);
```

Note that the overlay uses the same resize and input callback:

- For resize, you may or may not wish to share callbacks, depending on the desired functionality; for example, if you have a weathermap with annotations, both should resize in the same fashion.
- For input, the overlay usually sits on top of the normal window and receives the input events instead of the overlay window. Redirecting both to the same callback guarantees that you receive the events regardless of which window actually received them.
- The overlay has its own expose function: each time the overlay is exposed, it redraws itself.

5. Call **XRaiseWindow()** to make sure the overlay is on top of the window.

```
XRaiseWindow(dpy, XtWindow(ov_state.w));
```

### Overlay Troubleshooting

This section gives some advice on issues that can easily cause problems in a program using overlays:

- **Colormaps**. Overlays have their own colormaps. You therefore should call **XSetWMColormapWindows()** to create the colormap, populate it with colors, and to install it.

**Note:** Overlays on Silicon Graphics systems reserve pixel zero as the transparent pixel. If you attempt to create the colormap with AllocAll, the **XCreateColormap()** function will fail with a BadAlloc X protocol error. Instead of AllocAll, use AllocNone and allocate all the color cells except zero.

- **Window hierarchy**. Overlay windows are created like other windows, their parent window depends on what you pass in at window creation time. Overlay windows can be part of the same window hierarchy as normal windows and be children of the normal windows. An overlay and its parent window are handled as a single hierarchy for events like clipping, event distribution, and so on.
- **Color limitations**. On low-end Silicon Graphics systems, there are only a few overlay planes available; thus, items drawn in the overlay planes (such as menus) usually use only a few colors—no more than three colors and the transparent pixel in some cases. More recent low-end systems (24-bit Indy™ graphics), mid-range systems (Indigo2 IMPACT™), and high-end systems (RealityEngine™) support 8-bit overlay planes.
- **Input events.** The overlay window normally sits on top of the normal window. Thus, it gets all input events such as mouse and keyboard events. If the application is only waiting for events on the normal window, it will not get any of those events. It is necessary to select for events on the overlay window as well.
- **Not seeing the overlay**. Although overlay planes are conceptually considered to be “above” the normal plane, an overlay window can be below a normal window and thus clipped by it. When creating an overlay and a normal window, use **XRaiseWindow()** to ensure that the overlay window is on top of the normal window. If you use Xt, you must do this after the widget hierarchy has been realized.

## Rubber Banding

Rubber banding can be used for cases where it’s necessary to draw a few lines over a scene in response to a mouse movement. An example is the movable window outline that you get when resizing or moving a window. Rubber-banding is also used frequently by drawing programs.

The *4Dwm* window manager provides rubber banding for moving and resizing windows. However, if you need rubber banding features inside your application, you have to manage it yourself.

Here's the best way to do rubber banding with overlays (this is the method used by *4Dwm*, the default Silicon Graphics window manager):

1. Map an overlay window, with its *background* pixmap set to None (*background* is passed in as a parameter to **XCreateWindow()**). This window should be as large as the area over which rubber banding could take place.
2. Draw rubber bands in the new overlay window. Ignore resulting damage to other windows in the overlay plane.
3. Unmap the rubber-band window, which sends Expose events to other windows in the overlay plane.

### Using Popup Menus With the GLwMDrawingArea Widget

Pop-ups are used by many applications to allow user input. A sample program, *simple-popup.c*, is included in the source tree. It uses the function **XmCreateSimplePopupMenu()** to add a popup to a drawing area widget.

Note that if you are not careful when you create a popup menu as a child of GLwMDrawingArea widget, you may get a BadMatch X protocol error: The menu (like all other Xt shell widgets) inherits its default colormap and depth from the GLwMDrawingArea widget, but its default visual from the parent (root) window. Since the GLwMDrawingArea widget is normally not the default visual, the menu inherits a nondefault depth and colormap from the GLwMDrawingArea widget, but also inherits its visual from the root window (that is, inherits the default visual); leading to a BadMatch X protocol error. See "Inheritance Issues" on page 38 for more detail and for information on finding the error.

There are two ways to work around this:

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

**Example 4-1** Popup Code Fragment

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

    XmString button_labels[XtNumber(button_types)];

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

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

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

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

    /* specify visual directly */
```

```
    if (!(visinfo = glXChooseVisual(dpy, DefaultScreen(dpy), attribs)))
        XtAppError(app, "no suitable RGB visual");

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

    create_popup(frame);

    XtRealizeWidget(toplevel);

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

    XtAppMainLoop(app);
}
```

## Using Visuals

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

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

### Some Background on Visuals

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

Table 4-1 illustrates which X visuals support which type of OpenGL rendering, and whether the colormaps for those visuals are writable or not. Visuals that aren't available on Silicon Graphics systems are marked with an asterisk.

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

| OpenGL Rendering Mode | X visual | Writable colormap? |
|---|---|---|
| RGBA | TrueColor | no |
| RGBA | DirectColor* | yes |
| color index | PseudoColor | yes |
| color index | StaticColor* | no |
| not supported | GrayScale | yes |
| not supported | StaticGray | no |

* Not supported on Silicon Graphics systems.

An X server can provide multiple visuals, depending on the available hardware and software support. Each server has a default visual that can be specified when the server starts. You can determine the default visual with the Xlib macro **DefaultVisual()**.

Because you can't predict the configuration of every X server, and you may not know the system configuration your program will be used on, it is best to find out what visual classes are available on a case-by-case basis.

- From the command line, execute *xdpyinfo* for a list of all visuals the server supports.
- Execute *glxinfo* or *findvis* to find visuals that are capable of OpenGL rendering. The *findvis* command can actually look for available visuals with certain attributes. See the reference page for more information.
- From within your application, use the Xlib functions **XGetVisualInfo()** and **XMatchVisualInfo()**—or **glXGetConfig()**—or the GLX function **glXChooseVisual()**.

**Note:** For most applications, using OpenGL RGBA color mode and a TrueColor visual is recommended.

## Running OpenGL Applications Using a Single Visual

**Note:** This section applies only to IRIS IM.

In previous chapters, this guide has assumed separate visuals for the X and OpenGL portions of the program. The top-level windows and all parts of the application that aren't written in OpenGL use the default visual (typically 8-bit PseudoColor, but it depends on the configuration of the server). OpenGL runs in a single window that uses an Open GL visual.

An alternative approach is to run the whole application using an OpenGL visual. To do this, determine the suitable OpenGL visual (and colormap and pixel depth) at the start of the program and create the top-level window using that visual (and colormap and pixel depth). Other windows, including the OpenGL window, inherit the visual. When you use this approach, there is no need to use the GLwMDrawingArea widget; the standard IRIS IM XmDrawingArea works just as well.

The advantages of using a single visual include the following:

- **Simplicity**. Everything uses the same visual, so you don't have to worry about things like colormap installation more than once in the application. (However, if you use the GLwMDrawingArea widget, it does colormap installation for you—see "Drawing-Area Widget Setup and Creation" on page 29.)
- **Reduced colormap flashing**. Colormap flashing happens if several applications are running, each using its own colormap, and you exceed the system's capacity for installed hardware colormaps. Flashing is reduced for a single visual because the entire application uses a single colormap. The application can still cause other applications to flash, but all recent Silicon Graphics systems have multiple hardware colormaps to reduce flashing.
- **Easier mixing of OpenGL and X**. If you run in a single visual, you can render OpenGL to any window in the application, not just to a dedicated window. For example, you could create an XmDrawnButton and render OpenGL into it.

The advantages of using separate visuals for X and OpenGL include the following:

- **Consistent colors in the X visual**. If the OpenGL visual has a limited number of colors, you may want to allow more colors for X. For example, if you're using double buffering on an 8-bit machine, you have only 4 bitplanes (16 colors) per buffer. You can have OpenGL dither in such a circumstance to obtain approximations of other colors, but X won't dither, so if you're using the same visual for OpenGL and X, the X portion of your application will be limited to 16 colors as well.

This limiting of colors would be particularly unfortunate if your program uses the Silicon Graphics color-scheme system. While X chooses a color as close as possible to the requested color, the choice is usually noticeably different from the requested color. As a result, your application looks noticeably different from the other applications on the screen.

- **Memory savings**. The amount of memory used by a pixmap within the X server depends on the depth of the associated visual. Most applications use X pixmaps for shadows, pictures, and so on that are part of the user interface widgets. If you're using a 12-bit or 24-bit visual for OpenGL rendering and your program also uses X pixmaps, those pixmaps would use less memory in the default 8-bit visual than in the OpenGL visual.
- **Easier menu handling in IRIS IM**. If the top-level shell is not in the default visual, there will be inheritance problems during menu creation (see "Inheritance Issues" on page 38). You have to explicitly specify the visual depth and colormap when creating a menu. For cascading menus, specify depth and colormap separately for each pane.

## Using Colormaps

This section explains using colormaps in some detail. Note that in many cases, you won't need to worry about colormaps: Just use the drawing area widget and create a TrueColor visual for your RGBA OpenGL program. However, under certain circumstances, for example, if the OpenGL program uses indexed color, the information in this section is important. The section discusses these topics:

- "Background Information About Colormaps"
- "Choosing Which Colormap to Use"
- "Colormap Example"

## Background Information About Colormaps

OpenGL supports two rendering modes: RGBA mode and color index mode.

- In RGBA mode, color buffers store red, green, blue, and alpha components directly.
- In color-index mode, color buffers store indexes (names) of colors that are dereferenced by the display hardware. A color index represents a color by name rather than value. A colormap is a table of index-to-RGB mappings.

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

The X Window System supports six different types of visuals, with each type using a different type of colormap (see Table 4-1 on page 67). Although working with X colormaps may initially seem somewhat complicated, the X Window System does allow you a great deal of flexibility in choosing and allocating colormaps. Colormaps are discussed in detail and with example programs in Chapter 7, "Color," of O'Reilly Volume One.

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

### Color Variation Across Colormaps

The same index in different X colormaps does not necessarily represent the same color. Be sure you use the correct color index values for the colormap you are working with.

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

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

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

### Multiple Colormap Issues

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

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

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

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

**Example 4-2** Retrieving the Default Colormap for a Visual

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

    /* be lazy; using DirectColor too involved for this example */
    if (vi->class != TrueColor)
        fatalError("no support for non-TrueColor visual");
    /* if no standard colormap but TrueColor, make an unshared one */
```

```
    status = XmuLookupStandardColormap(dpy, vi->screen, vi->visualid,
        vi->depth, XA_RGB_DEFAULT_MAP,
        /* replace */ False, /* retain */ True);
    if (status == 1) {
        status = XGetRGBColormaps(dpy, RootWindow(dpy, vi->screen),
                                  &standardCmaps, &numCmaps,
                                  XA_RGB_DEFAULT_MAP);
        if (status == 1)
            for (i = 0; i < numCmaps; i++)
                if (standardCmaps[i].visualid == vi->visualid) {
                    cmap = standardCmaps[i].colormap;
                    XFree(standardCmaps);
                    return cmap;
                }
    }
    cmap = XCreateColormap(dpy, RootWindow(dpy, vi->screen),
        vi->visual, AllocNone);
    return cmap;
}
```

### Choosing Which Colormap to Use

When choosing which colormap to use, follow these heuristics:

1. First decide whether your program will use RGBA or color-index mode. Some operations, such as texturing and blending, are not supported in color index mode; others, such as lighting, work differently in the two modes. Because of that, RGBA rendering is usually the right choice. (See “Choosing between RGBA and Color-Index Mode” in Chapter 5, “Color,” of the *OpenGL Programming Guide*).

   OpenGL 1.0 and 1.1 and GLX 1.0, 1.1, and 1.2 require an RGBA mode program to use a TrueColor or DirectColor visual, and require a color index mode program to use a PseudoColor or StaticColor visual.

   **Note:** Remember that RGBA is usually the right choice for OpenGL on a Silicon Graphics system.

2. Choose a visual. If you intend to use RGBA mode, specify RGBA in the attribute list when calling **glXChooseVisual()**.

   If RGBA is not specified in the attribute list, **glXChooseVisual()** selects a PseudoColor visual to support color index mode (or a StaticColor visual if no PseudoColor visual is available).

If the framebuffer configuration extension is available, you can use a TrueColor or DirectColor visual in color index mode. See “The Framebuffer Configuration Extension” on page 171.

3. Create a colormap that can be used with the selected visual.
4. If a PseudoColor or DirectColor visual has been selected, initialize the colors in the colormap.

   **Note:** DirectColor visuals are not supported on Silicon Graphics systems. Colormaps for TrueColor and StaticColor visuals are not writable.

5. Make sure the colormap is installed. Depending on what approach you use, you may or may not have to install it yourself:
   - If you use the GLwMDrawingArea widget, the widget automatically calls **XSetWMColormapWindows()** when the GLwNinstallColormap resource is enabled.
   - The colormap of the top-level window is used if your whole application uses a single colormap. In that case, you have to make sure the colormap of the top level window supports OpenGL.
   - Call **XSetWMColormapWindows()** to ensure that the window manager knows about your window’s colormap. Here’s the function prototype for **XSetWMColormapWindows()**:

```
Status XSetWMColormapWindows(Display *display, Window w,
                             Window *colormap_windows, int count)
```

Many OpenGL applications use a 24-bit TrueColor visual (by specifying GLX_RGBA in the visual attribute list when choosing a visual). Colors usually look right in TrueColor, and some overhead is saved by not having to look up values in a table. On some systems, using 24-bit color can slow down the frame rate since more bits must be updated per pixel, but this isn’t usually a problem.

If you want to adjust or rearrange values in a colormap, you may have to use a PseudoColor visual, which has to be used with color-index mode unless the framebuffer configuration extension is available. Lighting and antialiasing are difficult in color-index mode, and texturing and accumulation don’t work at all. It may be easier to use double-buffering and redraw to produce a new differently-colored image, or use the overlay plane. In general, avoid using PseudoColor visuals if possible.

Overlays, which always have PseudoColor colormaps on current systems, are an exception to this.

### Colormap Example

Here's a brief example that demonstrates how to store colors into a given colormap cell:

```
XColor xc;
display = XOpenDisplay(0);
visual = glXChooseVisual(display, DefaultScreen(display),
                         attributeList);
context = glXCreateContext (display, visual, 0, GL_FALSE);
colorMap = XCreateColormap (display, RootWindow(display,
    visual->screen), visual->visual, AllocAll);
    ...
if (ind < visual->colormap_size) {
    xc.pixel = ind;
    xc.red = (unsigned short)(red * 65535.0 + 0.5);
    xc.green = (unsigned short)(green * 65535.0 + 0.5);
    xc.blue = (unsigned short)(blue * 65535.0 + 0.5);
    xc.flags = DoRed | DoGreen | DoBlue;
    XStoreColor (display, colorMap, &xc);
}
```

**Caution:** Do not use AllocAll on overlay visuals with transparency. If you do, **XCreateColormap()** fails because the transparent cell is read-only.

## Stereo Rendering

Silicon Graphics systems and OpenGL both support stereo rendering. In stereo rendering, the program displays a scene from two slightly different viewpoints to simulate stereoscopic vision, resulting in a 3D image to a user wearing a special viewing device. Various viewing devices exist; most of them cover one eye while the computer displays the image for the other eye, then cover the second eye while the computer displays the image for the first eye.

**Note:** Be sure to look at the *stereo* reference page for more information on stereo rendering (including sample code fragments and pointers to sample code).

In this section, you learn about

- "Stereo Rendering Background Information"
- "How to Do Stereo Rendering"

## Stereo Rendering Background Information

There are two basic approaches to stereo rendering, "Quad Buffer Stereo" and "Divided-Screen Stereo."

### Quad Buffer Stereo

Quad buffer stereo uses a separate buffer for the left and right eye, resulting in four buffers if the program is already using a front and back buffer for animation. Quad buffer stereo is supported on RealityEngine and Indigo2 Maximum IMPACT™ and will be supported on future high-end systems.

The main drawback of this approach is that it needs a substantial amount of framebuffer resources and is therefore feasible only on high-end systems. See "Performing Stereo Rendering on High-End Systems" on page 76 for step-by-step instructions.

### Divided-Screen Stereo

Divided-screen stereo divides the screen into left and right pixel lines. This approach is usually appropriate on low-end systems, which don't have enough memory for quad-buffer stereo.

If you put the monitor in stereo mode, you lose half of the screen's vertical resolution and pixels get a 1 x 2 aspect ratio. The XSGIvc extension does all X rendering in both parts of the screen. Note, however, that monoscopic OpenGL programs will look wrong if you use the extension.

When working with divided-screen stereo, keep in mind the following caveats:

- Since stereo is enabled and disabled without restarting the server, the advertised screen height is actually twice the height displayed.
- With quad-buffering, stereo pixels are square. If you're using divided-screen stereo, pixels are twice as high as they are wide. Thus, transformed primitives and images need an additional correction for pixel aspect ratio.

### For More Information

See the reference pages for the following functions: XSGIStereoQueryExtension(), XSGIStereoQueryVersion(), XSGIQueryStereoMode(), XSGISetStereoMode(), XSGISetStereoBuffer().

## How to Do Stereo Rendering

This section first explains how to do stereo rendering on high-end systems, then on low-end and mid-range systems.

### Performing Stereo Rendering on High-End Systems

To perform stereo rendering on high-end systems (RealityEngine, Indigo2 Maximum IMPACT, and future high-end systems), follow these steps:

1. Perform initialization, that is, make sure the GLX extension is supported and so on.
2. Put the monitor in stereo mode with the *setmon* command.
3. Choose a visual with front left, front right, back left, and back right buffers.
4. Perform all other setup operations illustrated in the examples in the previous two chapters: create a window, create a context, make the context current, and so on.
5. Start the event loop.
6. Draw the stereo image:

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

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

### Performing Stereo Rendering on Low-End and Mid-Range Systems

To perform stereo rendering on low-end and mid-range systems (including Indigo2 High IMPACT™), follow these steps:

1. Perform initialization, that is, make sure the GLX extension is supported and so on.
2. Put the monitor in stereo mode using the *setmon* command.
3. Call **XSGIStereoQueryExtension()** to see if the stereo extension is supported.
   - If stereo is not supported, exit.
   - If stereo is supported, call **XSGISetStereoMode()** to turn it on (options are STEREO_BOTTOM or STEREO_TOP).

4. Choose a visual with front left, front right, back left and back right buffers by calling **glXChooseVisual** with both GLX_DOUBLEBUFFER and GLX_STEREO in the attribute list.

5. Perform all other setup operations discussed in the examples in the previous two chapters: create a window, create a context, make the context current, and so on.

6. To draw the stereo image, use code similar to this pseudo-code fragment:

```
XSGISetStereoBuffer(STEREO_BUFFER_LEFT);
< draw left image >
XSGISetStereoBuffer(STEREO_BUFFER_RIGHT);
< draw right image >
glXSwapBuffers(...);
```

## Using Pixmaps

An OpenGL program can render to two kinds of drawables: windows and pixmaps. (Rendering to PBuffers is also possible if that extension is supported. See "The Pixel Buffer Extension" on page 179.) A pixmap is an offscreen rendering area. On Silicon Graphics systems, pixmap rendering is not hardware accelerated.

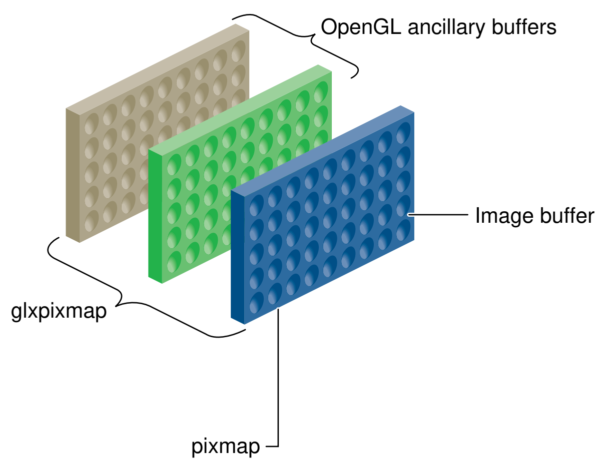


**Figure 4-2** X Pixmaps and GLX Pixmaps

In contrast to windows, where drawing has no effect if the window is not visible, a pixmap can be drawn to at any time since it resides in memory. Before the pixels in the pixmap become visible, they have to be copied into a visible window. The unaccelerated rendering for pixmap pixels has performance penalties.

This section explains how to create and use a pixmap and looks at some related issues:

- "Creating and Using Pixmaps" provides basic information about working with pixmaps.
- "Direct and Indirect Rendering" provides some background information; it is included here because rendering to pixmaps is always indirect.

### Creating and Using Pixmaps

Integrating an OpenGL program with a pixmap is very similar to integrating it with a window. It involves the steps given below. (Note that Steps 1-3 and Step 6 are discussed in detail in "Integrating Your OpenGL Program With IRIS IM" on page 15.)

1. Open the connection to the X server.
2. Choose a visual.
3. Create a rendering context with the chosen visual.

   This context must be indirect.
4. Create an X pixmap using **XCreatePixmap().**
5. Create a GLX pixmap using **glXCreateGLXPixmap().**

   ```
   GLXPixmap glXCreateGLXPixmap(Display *dpy, XVisualInfo *vis,
                                Pixmap pixmap)
   ```

   The GLX pixmap "wraps" the pixmap with ancillary buffers determined by *vis* (see Figure 4-2).

   The *pixmap* parameter must specify a pixmap that has the same depth as the visual that *vis* points to (as indicated by the visual's GLX_BUFFER_SIZE value), or a BadMatch X protocol error results.
6. Use **glXMakeCurrent()** to bind the pixmap to the context.

   You can now render into the GLX pixmap.

### Direct and Indirect Rendering

OpenGL rendering is done differently in different rendering contexts (and on different platforms).

- **Direct rendering** contexts support rendering directly from OpenGL via the hardware, bypassing X entirely. Direct rendering is much faster than indirect rendering, and all Silicon Graphics systems can do direct rendering to a window.
- In **indirect rendering** contexts, OpenGL calls are passed by GLX protocol to the X server, which does the actual rendering. Remote rendering has to be done indirectly; pixmap rendering is implemented to work only indirectly.

**Note:** As a rule, use direct rendering unless you're using pixmaps. If you ask for direct and your DISPLAY is remote, the library automatically switches to indirect rendering.

Here's some more information about indirect rendering:

In indirect rendering, OpenGL rendering commands are added to the GLX protocol stream, which in turn is part of the X protocol stream. Commands are encoded and sent to the X server. Upon receiving the commands, the X server decodes them and dispatches them to the GLX extension. Control is then given to the GLX process (via a context switch) so the rendering commands can be processed. The faster the graphics hardware, the higher the overhead from indirect rendering.

You can obtain maximum indirect-rendering speed by using display lists; they require a minimum of interaction with the X server. Unfortunately, not all applications can take full advantage of display lists; this is particularly a problem in applications using rapidly-changing scene structures. Display lists are efficient because they reside in the X server.

You may see multiple XSGI processes on your workstation when you are running indirect rendering OpenGL programs.

## Performance Considerations for X and OpenGL

Due to synchronization and context switching overhead, there is a possible performance hit for mixing OpenGL and X in the same window. GLX does not constrain the order in which OpenGL commands and X requests are executed. To ensure a particular order, use the GLX commands **glXWaitGL()** and **glXWaitX()**.

- **glXWaitGL()** prevents any subsequent X calls from executing until all pending OpenGL calls complete. When you use indirect rendering, this function does not contact the X server and is therefore more efficient than **glFinish().**
- **glXWaitX()**, when used with indirect rendering, is just the opposite: it makes sure that all pending X calls complete before any further OpenGL calls are made. This function, too, doesn't need to contact the X server, giving it an advantage over **XSync()** when rendering indirectly.
- Remember also to batch Expose events. See "Exposing a Window" on page 47.
- Make sure no additional Expose events are already queued after the current one. You can discard all but the last event.

## Portability

If you expect to port your program from X to other windowing systems (such as Windows NT), certain programming practices make porting easier. Here's a partial list:

- Isolate your windowing functions and calls from your rendering functions. The more modular your code is in this respect, the easier it is to switch to another windowing system.
- (For Windows NT porting only) Avoid naming variables with any variation of the words "near" and "far"—they are reserved words in Intel™ xx86 compilers. For instance, you should avoid the names _near, _far, __near, __far, near, far, Near, Far, NEAR, FAR, and so on.
- Windows NT programs by default have a small stack; don't allocate large arrays on the stack.
- Windows NT does not have an equivalent to **glXCopyContext().**