Getting Started with XGL Programming 3

This chapter explains how to compile an application program with XGL and how to make and run the XGL example programs. It also provides information on XGL concepts and programming and illustrates these concepts with an example program.

Compiling and Running XGL Programs

To compile an XGL application program, the application program source files must reference the XGL external C header file (used to define XGL data types and structures) with the statement `#include <xgl/xgl.h>` and link with the XGL library. If the application uses a toolkit, the application must also compile and link with toolkit libraries as described in the section below. If the application will be run on Sun hardware, it must also link with the OPEN LOOK graphics libraries.

Most applications will probably be compiled using `make(1)` to save time. When creating a `Makefile` for an XGL application, note the following:

- If the XGL files are not in the default location of `/opt/SUNWits/Graphics-sw/xgl`, the environment variable `$XGLHOME` must be set to point to the XGL files. This environment variable can be used in an XGL `Makefile`; it specifies the location of the XGL include files (using the -I option to `cc`) and the XGL library files (using the -L option to `cc` or `ld`).

Note – At the Solaris 2.5 release, the XGL `include` files do not reside in the same directory as the runtime library. However, a symbolic link from `/opt/SUNWits/Graphics-sw/xgl/include/xgl` to `/opt/SUNWsdk/sdk_2.5/xgl/include/xgl` is created when the SDK packages are installed. As a result, the `XGLHOME` environment variable works as in previous releases.

- Since the application will be running in the OpenWindows environment, the environment variable `$OPENWINHOME` should be set to point to the OpenWindows files in `/usr/openwin`. `$OPENWINHOME` is used the same way as `$XGLHOME` when compiling programs with the OpenWindows include files and libraries.
- Applications using the XView toolkit need to link with the XView library `-lxview` and the OPEN LOOK graphics library `-lolgx`. Applications using Xlib need to link with the X library `-lX11`. Applications using the OLIT widget set need to link with the OLIT library `-lXol` and with the X Toolkit Intrinsics library `-lXt`.

Note – Sun no longer supplies compilers as part of the standard operating system distribution. To compile an XGL application, you must have an ANSI-C compatible compiler that will run on the current release of Solaris.

The application is dynamically linked with the XGL library by including `-Bdynamic` in the link line. There is no static version of the XGL library in the XGL product. The following listing shows a sample `Makefile` for an XGL application:

```
# Sample Makefile for an XGL/OpenWindows application
.KEEP_STATE:
HFILES = <application .h files>
CFILES = <application .c files >

XGLHOMELIB:sh=echo ${XGLHOME-/opt/SUNWits/Graphics-sw/xgl}
OPENWINHOMELIB:sh=echo ${OPENWINHOMELIB-/usr/openwin}
OFILES = $(CFILES:%.c=%.o)
XGL_LIBS = -lxgl
SYS_LIBS = -lXol -lXt -lxview -lolgx -lX11 -ldga -ldl
OS_LIBS_5 = -lintl -lnsl -lsocket
OS_LIBS = $(OS_LIBS_$(OSVER))

LIBS = -L$(OPENWINHOMELIB)/lib -L$(XGLHOMELIB)/lib $(XGL_LIB) \
       $(SYS_LIBS) $(OS_LIBS) -lm
CFLAGS = -O
CPPFLAGS = -I$(XGLHOMELIB)/include -I$(OPENWINHOMELIB)/include

application: $(OFILES)
    $(LINK.c) $(OFILES) -R $(XGLHOMELIB)/lib:$(OPENWINHOMELIB)/lib \
    -Bdynamic $(LIBS) -lm -o application
```

Note that if either the `$XGLHOME` variable or the `$OPENWINHOME` variable is set, the `Makefile` will use that value. If these variables are not set (in other words, the XGL files and the OpenWindows libraries are in the default locations), the `Makefile` will use the default values.

Run-time Considerations

At runtime, the runtime loader needs to know the location of the XGL dynamic library `libxgl.so.3`. The application can pass this information to the loader either by linking the application at link time using the `-R` option or by setting the `LD_LIBRARY_PATH` environment variable to include the path to the XGL library. The link line of the sample `Makefile` above shows the use of the `-R` option. If a `Makefile` like the sample `Makefile` is used to compile an application, the `LD_LIBRARY_PATH` environment variable is not needed at runtime because the `Makefile` takes into account the values of `$XGLHOME` and `$OPENWINHOME`.

If the application was not compiled with the `-R` option, the user will need to set the `LD_LIBRARY_PATH` variable. The following example line from a user's `.cshrc` file or `.login` file shows the `LD_LIBRARY_PATH` variable being set to `$XGLHOME/lib` for the XGL library:

```
setenv LD_LIBRARY_PATH $OPENWINHOME/lib:$XGLHOME/lib
```

For more information on the `LD_LIBRARY_PATH` environment variable, see the `ld` (3) manual page.

Error Message Files and Font Files

The XGL runtime system looks in `$XGLHOME/lib/locale/en_US/LC_MESSAGES` for translated error message files and device error message files. It looks in `$XGLHOME/lib/xglfonts/stroke` for its stroke font files. If `$XGLHOME` is not set, the runtime system looks in `/opt/SUNWits/Graphics-sw/xgl/lib/locale/en_US/LC_MESSAGES` for the error files and in `/opt/SUNWits/Graphics-sw/xgl/lib/xglfonts/stroke` for the stroke font files. The search path for the font files can be changed by the application. See page 50 for information.

XGL Loadable Pipelines

XGL renders images via loadable pipelines. The loadable pipelines are shared object library modules containing device-specific functions. Although `libxgl` is loaded by the runtime loader, the loadable pipeline shared object files are loaded explicitly by a call in the XGL code. The loadable pipelines are located in `$XGLHOME/lib/pipelines` if they have been installed. If `$XGLHOME` is not set, then installed pipelines are assumed to be in the default area of `/opt/SUNWits/Graphics-sw/xgl/lib/pipelines`.

Note – If the application is running remotely and the server has loaded the PEX extension, `XGLHOME` is not used to load the device pipelines, but it is used to load the font files and error message files.

Using `make(1)` To Build XGL Example Programs

The Solaris SDK CD-ROM contains XGL example programs that use the XGL library to illustrate basic XGL concepts. These programs are described throughout this manual. The example programs are installed in `/opt/SUNWsdk/sdk_2.5/xgl/demo/examples`. This directory also contains a `Makefile` for compiling the programs. As mentioned previously, the environment variables `$OPENWINHOME` and `$XGLHOME` must be set before compiling or running these programs if the files are not in the default locations. Note that since XGL has been designed to run in the OpenWindows windowing environment, all the example programs in this manual run within this environment.

The following example shows how to create and run the XGL example program `hello_world`.

```
example% cd /opt/SUNWsdk/sdk_2.5/xgl/demo/examples
example% make hello_world
example% hello_world
```

To make all the example programs at the same time, type `make all`.

Basic XGL Concepts

The XGL programming model and the relationship between XGL and the X window system were briefly introduced in Chapter 1. The sections that follow provide additional information on XGL and the X window system, and describe more fully the XGL object-based programming model. For an example of an XGL program, turn to "Example Program" on page 37.

XGL and the X Window System Environment

An XGL application is an X client program running in an X window system environment, such as Sun's OpenWindows environment. The X window system is composed of several components. The central component is the display server, which is responsible for managing output requests from client applications to draw onto the display and for distributing input events to the appropriate client applications. The server and the client applications communicate via the X and/or PEX protocol(s). During a work session, client applications generate protocol queries and requests using the Xlib or PEXlib programming interface or an X toolkit, and the server processes these requests.

An XGL application program uses Xlib or X toolkit functions to connect to the server, create a window on the display, and handle event interpreting. The server manages the relationship among the various windows on the screen and provides input and events to the application, and the toolkit provides the application's user interface. The application renders its geometric data to the window using XGL primitives. The Xlib or X toolkit calls coexist with the XGL calls, sharing the same process space and drawable area on the screen. Figure 3-1 illustrates a high-level view of this relationship.

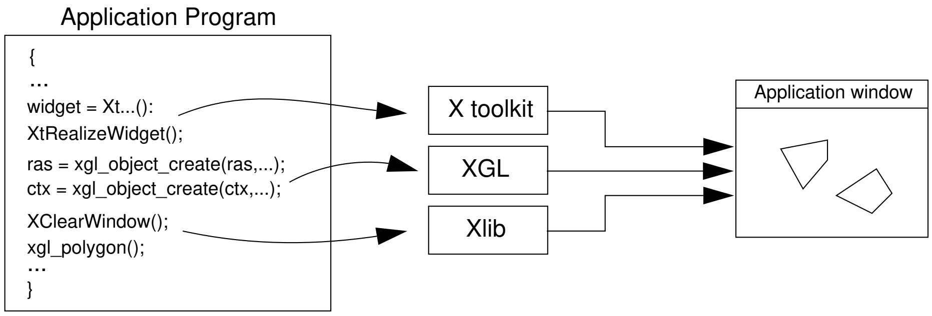

Figure 3-1 High-level View of an XGL Application Program

Rendering Graphics Locally and Remotely

When a client XGL application is running locally under Sun's OpenWindows server and the hardware supports Direct Graphics Access (DGA) protocol, XGL renders using DGA technology rather than rendering through the server via the X protocol. DGA is a set of mechanisms that enables OpenWindows client processes to directly drive a graphics device that is under the control of the OpenWindows server. Although the server manages the resources of the device, the XGL client process coordinates with the server to send rendering commands directly to the device rather than sending X11 protocol messages to the server. This allows XGL to eliminate the communication overhead of the server-based window system, thereby improving performance.

Figure 3-2 on page 29 illustrates the XGL system's relationship to DGA and the server. When an application program creates a Raster Device object, the XGL library automatically determines whether DGA is available and whether XGL supports the frame buffer. If DGA is available and the frame buffer includes DGA support, XGL synchronizes with the server to share direct access to the frame buffer. In this case, XGL renders graphics geometry to the frame buffer, and the server performs other window operations. Note that XGL will render through Xlib or PEXlib when used with a non-OpenWindows server. This allows XGL to render to any X11 server; however, rendering will be accelerated on a machine running the OpenWindows server.

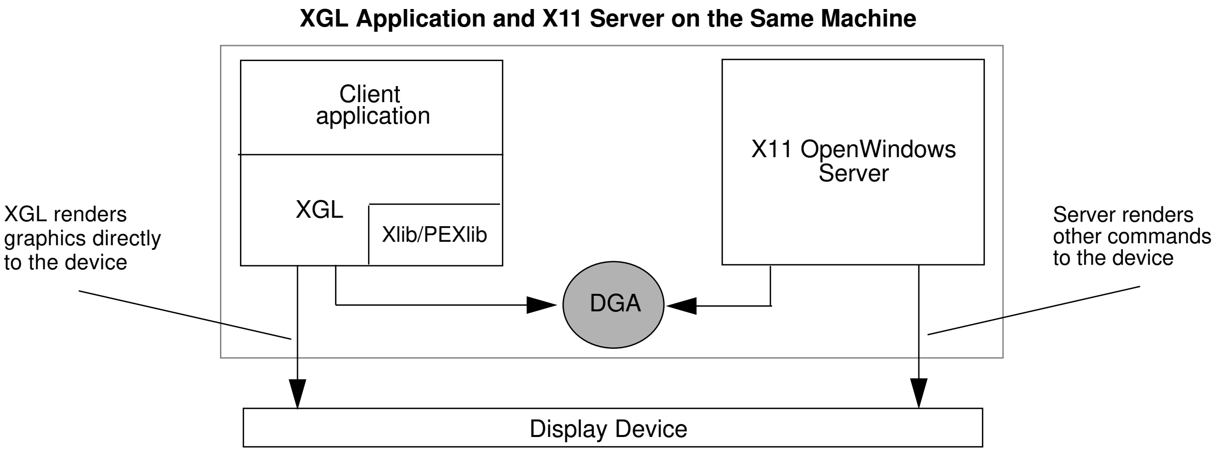

Figure 3-2 Using DGA to Render Locally

Like all X applications, an XGL application can be run remotely and displayed on a user's local workstation if the two workstations are part of a network. Remote rendering through the X11 protocol or the PEX protocol is handled automatically by XGL. When the XGL client program is running remotely, XGL uses Xlib or PEXlib to do all rendering. If the server includes the PEX extension and XGL has access to its PEX loadable library, XGL uses PEXlib to render. If PEX is not available, XGL uses Xlib for 2D rendering. For primitives and rendering options that are not supported by Xlib, XGL does scan conversion to send the pixels through Xlib to the device. Figure 3-3 on page 30 illustrates remote rendering.

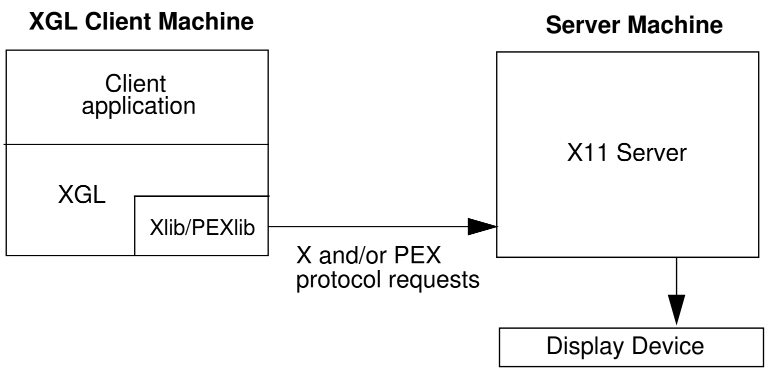

Figure 3-3 XGL and Remote Rendering

How an XGL Application Works

As mentioned above, an XGL application follows the general format of Xlib, PEXlib, or X toolkit programming, using the event-driven model of X applications for interaction handling. With Xlib calls, PEXlib calls, or X toolkit calls, the application first creates the window system objects that it needs, such as a window for graphics display, and then opens the XGL system. When XGL is opened, it automatically creates the System State object as well as internal objects that handle interactions between the device-dependent and the device-independent parts of the XGL system. To set up a framework for rendering, the application must create a Device object, which is an abstraction representing the display device, and a Context object, which controls all rendering actions on a device. These objects are created using an XGL object creation operator that takes attributes describing the object as input arguments, creates an instance of the object, and returns a handle to the object. The application program must associate the Device object with the Context object before geometry can be rendered. When the Context object and Device object are associated, the application can use XGL primitives to render geometry and can pass control of the program to Xlib or an X toolkit to process events.

During the work session, the application can change Context attributes to change the display characteristics of geometry. It can also render geometric data using XGL drawing primitives and create other objects as needed. For example, the application might want to create several Stroke Font objects to enable the use of different character sets, or create Line Pattern objects to provide the user with application-specific line patterns. Multiple Device objects, such as Window Raster and Memory Raster devices, can be associated with and disassociated from the Context object as needed for rendering. When the application exits XGL, the System State object destroys existing objects, frees resources, and then destroys itself, closing XGL.

XGL Drawing Primitives

The XGL library provides a set of drawing primitives that the application can use to render geometric data. The available primitives include basic line, polygon, and text primitives as well as more complex primitives, such as NURBS curve, curved surface, and quadrilateral mesh primitives. In the XGL system, geometry is rendered via the Context object, which also maintains graphic and environment state information that is used in rendering. Because XGL is an immediate mode system, it needs access to current state information at the time of rendering. Since the application can create more than one Context object and can render the same geometric data using different Context objects, the application must define which Context object is to be used to render. Thus, the drawing primitives require the Context object as an input parameter and are considered to belong to the Context object.

The drawing primitives are listed in Table 3-1 and are described in more detail in Chapter 8, “Primitives and Graphics Context Attributes”.

Table 3-1 XGL Drawing Primitives

Primitive Type	**Description**	**XGL Primitive**
Lines	A set of unconnected lines	`xgl_multipolyline()`
B-spline curves	A spline (NURBS) curve	`xgl_nurbs_curve()`
Markers	A set of markers	`xgl_multimarker()`
Circles	A set of circles	`xgl_multicircle()`
Circular arcs	A set of arcs	`xgl_multiarc()`
Elliptical arcs	A set of 3D elliptical arcs	`xgl_multi_elliptical_arc()`
Rectangles	A set of rectangles	`xgl_multirectangle()`
Polygon	A single planar polygon	`xgl_polygon()`
Multiple polygons	A set of 3- or 4-sided polygons	`xgl_multi_simple_polygon()`
Stroke text	A text string	`xgl_stroke_text()`

Table 3-1 XGL Drawing Primitives

Primitive Type	Description	XGL Primitive
Annotation text	A text string displayed in a plane parallel to the display surface	`xgl_annotation_text()`
Quadrilateral mesh	A set of connected quadrilateral polygons	`xgl_quadrilateral_mesh()`
Triangle strip	A set of connected triangular polygons	`xgl_triangle_strip()`
Triangle list	A set of connected triangles arranged as a triangle strip or as a triangle star, or a set of unconnected triangles	`xgl_triangle_list()`
Curved surface	B-spline surface	`xgl_nurbs_surface()`

Handling 2D and 3D Data

XGL handles the dimensionality of application data via the dimension of the Context object, the type of point data input to the drawing primitives, and the set of attributes available to a Context. Context objects can be either 2D or 3D. An application that needs to render both 2D and 3D model data will typically create two Context objects, one for 2D rendering and one for 3D rendering. When the Context object is created, a default rendering pipeline, which includes transformations and clipping,, is also created. Creation of a 2D Context object results in the creation of a pipeline tailored to the processing of 2D geometry; creation of a 3D Context results in the creation of a more complex 3D pipeline, which handles lighting, shading, and texturing.

Application model data must be input to the drawing primitives using a point type appropriate to the Context object. Thus, for example, a 2D application might define a 2D Context object and set up the application data in XGL 2D point structures, choosing from integer or floating point structures. The XGL library provides a wide variety of point data types, including point types with and without color and/or normal data.

The dimension of the Context object also controls the set of attributes available to the application for rendering. For example, a 2D Context object includes a set of attributes that controls how the front of a surface is displayed; a 3D Context object has access to the front surface attributes but has an additional set of attributes for back surface rendering as well.

Once the appropriate Context object has been created, the application model data has been set up in XGL point data structures, and the applicable attributes have been set, the application can render using the drawing primitives. Many of the XGL drawing primitives can be used for both 2D and 3D rendering, but some are specific to 3D geometry, as shown in Table 3-2.

Table 3-2 Dimensionality of XGL Primitives

Primitive	2D Rendering	3D Rendering
`xgl_multipolyline()`	✓	✓
`xgl_multimarker()`	✓	✓
`xgl_multicircle()`	✓	✓
`xgl_multiarc()`	✓	✓
`xgl_multi_elliptical_arc()`		✓
`xgl_multirectangle()`	✓	✓
`xgl_polygon()`	✓	✓
`xgl_multi_simple_polygon()`	✓	✓
`xgl_stroke_text()`	✓	✓
`xgl_annotation_text()`	✓	✓
`xgl_quadrilateral_mesh()`		✓
`xgl_triangle_strip()`		✓
`xgl_triangle_list()`		✓
`xgl_nurbs_curve()`	✓	✓
`xgl_nurbs_surface()`		✓

3

More on XGL Object-Based Programming

XGL represents graphics information with abstract data structures called objects. An XGL object represents a graphics resource, such as a line pattern or a font. Each object describes a virtual component of the graphics rendering system and contains information necessary to perform graphics operations. A display device, for example, is known to the XGL programmer as a Window Raster Device object. This abstraction of the display device hides the specifics of the device-dependent code for the graphics hardware device and lets the XGL programmer interact with the device in a device-independent manner.

Internally, XGL objects are instances of classes. A class is an abstract data type that combines state information with functions that perform actions on the object. The class's attributes and functions define the characteristics of the object. A class is only a data type definition; it does not exist in memory. A class must be instantiated to be used, and it can be instantiated many times to create distinct objects that all belong to the same class. Objects exist in system memory and provide the functionality that the application program needs to render geometry.

Figure 3-4 on page 35 shows a diagram of the XGL class hierarchy with objects that have been instantiated by calls to `xgl_object_create()`. The illustration shows a possible XGL runtime system consisting of two Context objects, a Window Raster object, a Memory Raster object, two Stroke Font objects, and a Transform object.

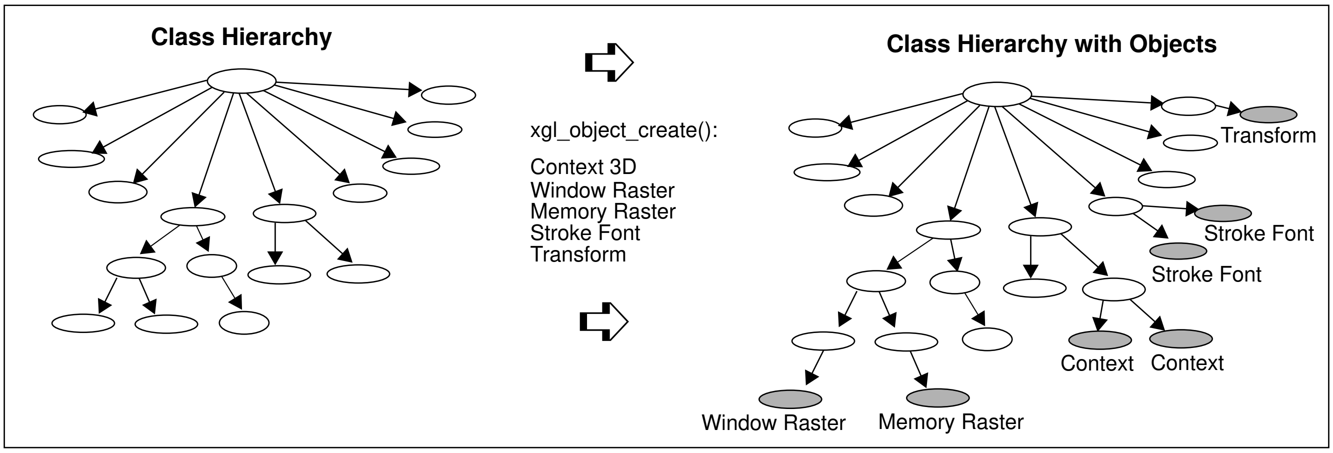

Figure 3-4 Instantiated Objects

Operators

Operators are functions that control the behavior of an object. Most XGL objects have a set of operators that handle operations required by the object. For example, the Transform object includes operators that perform transformation operations, such as matrix concatenation. The XGL system also provides operators that perform functions common to objects, such as object creation or destruction. These operators, listed in Table 3-3, provide a consistent method of working with all XGL objects.

Table 3-3 Generic XGL Operators

Operator	Description
`xgl_object_create()`	Creates an XGL object.
`xgl_object_set()`	Sets the value of an attribute.
`xgl_object_get()`	Gets the value of an attribute.
`xgl_object_destroy()`	Destroys an object.

Operators provide a way for the application to access an object and manipulate the object's data structures, since direct access to the object data structures is not provided. If, for example, the application program needs to change the color of a line, it must use the `xgl_object_set()` operator to change the value of the line color attribute, since it cannot directly access the line color attribute field in the object data structure.

Attributes

Attributes control much of the functionality of XGL, such as the appearance of rendered geometry, the way picking is handled, and the orientation of virtual device coordinate space. Although some attributes are read-only, an application can change the value of most attributes at any time. Attributes are input as arguments to XGL operators, some of which take an attribute-value list as one of the input parameters. Each attribute-value pair in the list consists of an attribute name followed by its value. The attribute-value list may have as few as zero attribute-value pairs or as many as the total number of the object's attributes, but in all cases it is terminated with the `NULL` character, where `NULL` is defined as in `stdio.h`.

Object Relationships

Some objects use other objects as resources. When an association has been established between two objects, the objects communicate via messages that inform the using object of changes that have occurred in the data structures of the used object. For example, if an application has created a new Line Pattern object and has associated the Line Pattern object with the Context object, the Context object will automatically be kept up to date on the characteristics of the Line Pattern object so that the line pattern will be rendered correctly.

Object relationships are set up by the application and are managed internally by XGL object management functions. When an application has created a new object, it establishes the association between the new object and an existing object using the `xgl_object_set()` operator, a connecting attribute, and the handles of the two objects. Table 3-4 lists objects that are associated as graphics resources with the Context and Device objects.

Table 3-4 Object Relationships

Using object	**Object being used**
Raster Device	System State Color Map
Context	System State Device Data Map Texture Texture Map Transform Line Pattern Marker MipMap Texture Pcache Stroke Font Light (3D Context only)

Information on the specific attributes that associate resource objects with the Context or Device objects is available in the chapters that follow.

Example Program

The `hello_world.c` program is a simple XGL program using the OLIT toolkit. The program opens XGL, creates a Window Raster Device object, renders simple text on the Raster in the default font, and closes XGL. The program's `main()` function follows these general steps:

- Initialize the OLIT toolkit and the X Toolkit internals, opening the connection to the server.
- Create the toplevel shell and the pane widget that XGL will render into, registering a callback function for expose events on the pane widget.
- Realize the widgets.
- Initialize XGL, creating the System State object.
- Create a Window Raster Device object associated with the pane.

- Create a 2D Context object and associate the Raster object with the Context object.
- Start the Xt main event loop. Once the main event loop has started, Xt handles program execution.

The output of `hello_world.c` is shown in Figure 3-5, and the complete program listing follows. Turn to page 41 for a more detailed discussion of the program. The code for this program is located in `/opt/SUNWsdk/sdk_2.5/xgl/demo/examples`, with the other XGL sample programs. To compile the program, type `make hello_world` in the example programs directory.

Figure 3-5 Output of `hello_world.c`

Code Example 3-1 A First Example Program

```
/*
 * hello_world.c
 */

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <X11/Shell.h>
#include <Xol/OpenLook.h>

#include <Xol/DrawArea.h>

#include <xgl/xgl.h>
```

```
/* Global variables */
static Xgl_object       ctx = NULL;     /* XGL context object */
static Xgl_object       sys_state;
static Xgl_object       win_ras = NULL; /* XGL window raster */

main (
    int                 argc,
    char                *argv[])
{
    static void         redraw();

    XtAppContext        app;

    Widget              toplevel, pane;

    Display             *display;     /* pointer to X display */
    Window              xwindow;      /* XID of window */
    Xgl_X_window        xgl_x_win;    /* XGL-X data structure */
   Xgl_obj_desc       win_desc;     /* XGL window raster structure */

    OlToolkitInitialize((XtPointer)NULL);
    toplevel = XtVaAppInitialize(&app, "XGL Hello",
        (XrmOptionDescList)NULL, 0,
        &argc, argv, (String *)NULL,
        XtNtitle,        "XGL Hello World Program",
        XtVaTypedArg, XtNvisual, XtRString,
        "PseudoColor", sizeof("PseudoColor"),
        NULL);

    pane = XtVaCreateManagedWidget( "pane",
        drawAreaWidgetClass,    toplevel,
        XtNborderWidth,         1,
        XtNwidth,               500,
        XtNheight,              200,
        NULL);
    XtAddCallback(pane, XtNexposeCallback, redraw, NULL);

    XtRealizeWidget(toplevel);

 /* get X stuff */
    display = (Display *) XtDisplay (pane);
    xwindow = XtWindow (pane);

    /* put X stuff into XGL data structure */
    xgl_x_win.X_display = (void *) display;
    xgl_x_win.X_window  = (Xgl_usgn32) xwindow;
```

```
    xgl_x_win.X_screen  = (int) DefaultScreen (display);

/*
 *   Open XGL, create a Raster and a Context.
 *   Attach the raster as the Context's Device.
 *   Initialize the context text attributes to render the text.
 */
    sys_state = xgl_open (NULL);

    win_desc.win_ras.type = XGL_WIN_X;
    win_desc.win_ras.desc = &xgl_x_win;
    win_ras = xgl_object_create (sys_state, XGL_WIN_RAS,
                            &win_desc, NULL);

    /* create XGL graphics Context object using the
     * Window Raster object */
    ctx = xgl_object_create (sys_state, XGL_2D_CTX, NULL,
                       XGL_CTX_DEVICE, win_ras,
                       XGL_CTX_DEFERRAL_MODE,XGL_DEFER_ASAP,
                       XGL_CTX_STEXT_CHAR_HEIGHT, 20.0,
                       NULL);

     XtAppMainLoop(app);
}

static void
redraw()
{
    static Xgl_pt_f2d   text_pos; /* start pos of text in window */

    text_pos.x = 90.0;
    text_pos.y = 100.0;

    if (ctx) {
        xgl_window_raster_resize(win_ras);
        /* clear the display */
        xgl_context_new_frame (ctx);

        /* draw the stroke text */
        xgl_stroke_text (ctx, "Hello XGL World", &text_pos, NULL);
    }
}
```

Creating the Window and Registering Callback Procedures

The `main()` routine of `hello_world.c` initializes the OLIT toolkit, the Xt toolkit, and XGL, and opens the connection to the server. It creates the OLIT widgets that the application needs, in this case, a DrawArea widget for the pane, registers a callback procedure to handle expose events on the pane, and creates a window for the widget. It also creates the XGL objects that the application needs. Finally, it starts the main event loop to wait for X events.

The program creates a drawing surface 500 units wide and 200 units high. The window frame is labeled "XGL Hello World Program". The visual type of the window is PseudoColor.

Opening XGL

The `xgl_open()` call initializes the XGL environment.

```
sys_state = xgl_open(NULL);
```

The `xgl_open()` operator creates and returns the System State object. The System State object handles the creation of other XGL objects in subsequent `xgl_object_create()` calls.

Creating a Window Raster Device Object

To create a Window Raster Device object, the application must provide XGL with information about the X11 window with which the Raster object is associated. The application must get a pointer to the display, the X handle to the window, and an integer value identifying the screen that XGL will render to. This information is stored in an XGL window descriptor data structure that is passed to `xgl_object_create()` when the Window Raster object is created. The following code fragments show the Xt calls that retrieve window information, the XGL window data structure, and the `xgl_object_create()` call.

```
/* get X stuff */
   display = (Display *) XtDisplay(pane);
   xwindow = XtWindow (pane);
          .
          .
          .
   /* put X stuff into XGL data structure */
```

```
    xgl_x_win.X_display = (void *) display;
    xgl_x_win.X_window = (Xgl_usgn32) xwindow;
    xgl_x_win.X_screen = (int) DefaultScreen(display);
          .
          .
          .
    win_desc.win_ras.type = XGL_WIN_X;
    win_desc.win_ras.desc = &xgl_x_win;
    win_ras = xgl_object_create(sys_state, XGL_WIN_RAS, &win_desc,
                                 NULL);
```

The `xgl_object_create()` operator returns a handle to the Window Raster Device object. The System State object stores a pointer to the Device object in its list of API objects.

Creating a Context Object

The Context object keeps track of attribute information for the XGL primitives. In this example, a 2D Context object is created by a call to `xgl_object_create()` with a *type* value of `XGL_2D_CTX`.

```
ctx = xgl_object_create(sys_state, XGL_2D_CTX, NULL,
                        XGL_CTX_DEVICE, win_ras,
                        XGL_CTX_DEFERRAL_MODE, XGL_DEFER_ASAP,
                        XGL_CTX_STEXT_CHAR_HEIGHT, 20.0,
                        NULL);
```

The Context's deferral mode is set so that XGL will render a primitive as soon as it is received. The character height for the stroke font is set to 20.0 units. The previously created Device object is associated with the Context object with the `XGL_CTX_DEVICE` attribute and the Device object's handle. The newly created Context object is registered in the System State object passed to the `xgl_object_create()` operator.

Rendering Geometry

The `redraw`() procedure renders the geometry data that the application supplies. In this program, the `redraw`() procedure clears the display using the XGL operator `xgl_context_new_frame()` and then draws text into the window with the primitive `xgl_stroke_text()`. The `xgl_stroke_text()` primitive renders the string “Hello XGL World” to the text position, using the text attributes in the Context object.

```
static void
redraw()
{
    static Xgl_pt_f2dtext_pos;/* start pos of text in window */

    text_pos.x = 90.0;
    text_pos.y = 100.0;

    if (ctx) {
        xgl_window_raster_resize(win_ras);
        /* clear the display */
        xgl_context_new_frame (ctx);

       /* draw the stroke text */
        xgl_stroke_text (ctx, "Hello XGL World", &text_pos, NULL);
    }
}
```

Objects Provided at XGL Initialization

A set of XGL objects is created during the initialization of the XGL library. Although the application must create the Context object and the Device object, other objects are automatically created, as follows:

- The System State object is automatically created during the `xgl_open()` call. The System State object in turn creates the following objects:
  - Line Pattern objects that supply the application with a set of predefined line patterns. These predefined Line Pattern objects cannot be modified by the application. To create additional Line Patterns, the application must create additional Line Pattern objects.
  - Marker objects that provide predefined symbols. Like the predefined Line Pattern objects, these Marker objects cannot be modified by the application. To create additional markers, the application must create additional Marker objects.
- When the application creates the Device object, the XGL system creates a default two-color (black and white) Color Map object. This Color Map cannot be modified. To render in color, the application must create its own Color Map object and associate it with the Device object.

- When the application creates a Context object, the following objects are also created:
  - Transform objects for transformation matrices used by the viewing pipeline. These Transform objects are the Local Model Transform, the Global Model Transform, the Model Transform, the View Transform, the VDC Transform, and for 3D Contexts, the Normal Transform. These Transforms are set to identity at initialization, but the application can retrieve several of the matrices and manipulate the values using the Transform operators. Applications can also create new Transform objects as necessary.
  - A Stroke Font object that defines the Roman_M font in the XGL font directory as the default font. To use different font, the application must create a new Stroke Font object, set it to the appropriate font name, and associate it with the Context object.

Note that no Light objects are created at initialization, but Context environment attributes can create an array of lights whose values can be retrieved and set.

Programming Tips

This section provides tips on writing XGL programs. The first group of tips contains general information and considerations about writing XGL programs. The second group of tips contains information about using XView/OLIT and XGL. Finally, XGL limitations are briefly listed.

General Tips

- The default value of the Context attribute `XGL_CTX_DEFERRAL_MODE` is `XGL_DEFER_ASTI` ("At Some TIme"). In this mode, the application must call `xgl_context_post()` to ensure that all the data is flushed from the XGL internal buffer and displayed.
- Assuming that an application has obtained X information on the X11 display, the frame window containing the drawing window, and the canvas window that XGL will draw to, the application can use the following lines of code to enable the window manager to install an application color map for the canvas window.

```
Atom      catom;
catom = XInternAtom(display, "WM_COLORMAP_WINDOWS", False);
XChangeProperty(display, frame_window, catom, XA_WINDOW, 32,
               PropModeAppend, &canvas_window, 1);
```

- `xgl_close()` should be called to ensure that all resources are released.

XView Tips

- The XView attribute `CANVAS_AUTO_CLEAR` should be set to `FALSE`, since the XGL `xgl_context_new_frame()` operator can clear the canvas (using DGA) faster than Xlib can. `xgl_context_new_frame()` also performs other actions, such as clearing the Z-buffer.
- The application should supply an event procedure to handle resize and repaint events for the canvas paint window of the XView canvas. When a `WIN_RESIZE` event occurs, the XGL `xgl_window_raster_resize()` operator should be called. If there is more than one XGL window raster in an application, the application must determine which *Xgl_win_ras* object corresponds to the *Xv_window* object passed to the event procedure.

  In addition, before calling `xgl_window_raster_resize()`, the application should call `XSync()` if it will be running as a DGA application. The XGL call `xgl_window_raster_resize()` uses DGA to grab window size information. Because of the way XView handles geometry management, the window size returned is the previous window size. Calling `XSync()` before `xgl_window_raster_resize()` will synchronize the server and XGL.
- The XView attribute `CANVAS_FIXED_IMAGE` should be set to `FALSE` so that the server will inform the application, via the `CANVAS_RESIZE_PROC`, that the canvas has been resized. When this attribute is `TRUE`, the `CANVAS_RESIZE_PROC` is not called when the canvas is made smaller.
- If the application defines a color map for a canvas, the application must set the `WM_COLORMAP_WINDOWS` property on the its top-level window to tell the window manager which canvas will use the color map. The following lines of code shows how to accomplish this under XView.

```
Atom      catom;
/* These four lines get X info from XView objects */
display = (Display *)xv_get(frame, XV_DISPLAY);
```

```
frame_window = (Window)xv_get(frame, XV_XID);
canvas_window = (Window)xv_get
                (canvas_paint_window(canvas),XV_XID);
catom = XInternAtom(display, "WM_COLORMAP_WINDOWS", False);
XChangeProperty(display, frame_window, catom, XA_WINDOW, 32,
            PropModeAppend, &canvas_window, 1);
```

Note that an application does not need to set the `WM_COLORMAP_WINDOWS` property if it is only using one color map.

- Instead of calling `xgl_close()` after `xv_main_loop` returns, the application should call `xgl_close()` in a routine interposed on the quit event.

OLIT Tips

- The application should call `xgl_close()` from the "Quit" selection in the window menu. This is done by adding a callback to the toplevel shell:

```
OlAddCallback(toplevel, XtNwmProtocol, WMCallback, NULL);
```

The following lines of code show an example of a procedure that could be called when the shell receives a `WM_PROTOCOL` message.

```
void
WMCallback(widget, clientdata, calldata)
    Widget          widget;
    XtPointer       clientdata;
    XtPointer       calldata;
{
    OlWMProtocolVerify *wmvrfy = (OlWMProtocolVerify *)calldata;

    if ((wmvrfy->msgtype == OL_WM_DELETE_WINDOW) &&
                                (application wants to stay up))
        return;
    else {
        xgl_close(sys_state);
        OlWMProtocolAvction(widget, wmvrfy, OL_DEFAULTACTION);
    }
}
```

XGL Limitations

When you are writing XGL programs, note the following limitations:

- No Light objects are created at XGL initialization. This means that an XGL program does not have any lights automatically available. You can make lights available by creating a Light object or by setting the Context attributes `XGL_2D_CTX_LIGHT_NUM` and `XGL_3D_CTX_LIGHTS`. For more information on lighting, see "Lighting, Shading, and Depth Cueing" on page 403.
- Turning on error checking using the attribute `XGL_SYS_ST_ERROR_DETECTION` may affect performance. A good approach may be to set `XGL_SYS_ST_ERROR_DETECTION` to `TRUE` while you are developing your application and set it to `FALSE` when you compile your application for distribution. For more information on error checking, see "Error Detection and Reporting" on page 51.
- The concurrent use of backing store and double buffering is not supported. XGL enables whichever is requested first. For more information on backing store and double buffering, see page 69.
- Backing store (`XGL_WIN_RAS_BACKING_STORE`) should not be enabled when accumulation is needed or when `XGL_3D_CTX_HLHSR_MODE` is set to `XGL_HLHSR_Z_BUFFER` for a 3D Context. The reason for this is that the X server does not handle the copying of a Z-buffer or accumulation buffer for backing store. For more information, see page 70.
- Although you can mix XGL and Xlib calls, take care when mixing these calls because latencies involved with XGL communication with graphics hardware and Xlib communication with the server can result in geometry being drawn in an unexpected order. If the order in which the primitives are drawn is important to your application, you should take following precautions when mixing XGL and Xlib functions:
  - When the application has finished drawing with XGL primitives, call `xgl_context_flush(ctx,XGL_FLUSH_SYNCHRONIZE)` to ensure that XGL primitives are displayed before calling Xlib functions. This operator should be called even if the `XGL_CTX_DEFERRAL_MODE` is `XGL_DEFER_ASAP`.
  - Xlib calls should be flushed with `XSync` before further XGL commands are issued. However, the use of `XSync` may cause some performance degradation. Note also that when rendering and clearing the window with

Xlib calls, the application must clear the window with `xgl_context_new_frame()` before rendering with XGL primitives, even if the window has previously been cleared with Xlib calls.

- Your application must open XGL with a call to `xgl_open()` before it can call `xgl_inquire()`. Device characteristics cannot be queried before XGL is open. For information on `xgl_inquire()`. see page 92.
- Accumulation buffer functionality is currently only available on RGB rasters. For information on accumulation buffers, see page 228.
- Texturing is only available on 3D RGB rasters. For information on texture mapping, see “Texture Mapping” on page 457.