Graphics 5

Introduction

With Solaris VISUAL, SunSoft is committed to providing a means to its OEMs and independent software vendors (ISVs) to take advantage of various graphics and imaging libraries such as Display PostScript® (DPS), XGL™, XIL, PEXlib, Xlib on both SPARC® and x86 platforms, and on third-party graphics peripherals which become accessible devices under Solaris 2.x.

By opening up the device layers, the independent hardware vendors (IHVs) are able to provide greater access to third-party display devices. Solaris 2.x focuses on foundation layers of graphics libraries, enabling more third-party vendors to offer more choices when developing new applications or application program interfaces (APIs).

This chapter talks about the major components of Solaris VISUAL, as well as migration issues of moving Solaris 1.x graphical applications that exist on SPARC platforms today onto the less platform-specific Solaris 2.x.

Overview

In early versions of SunOS, developers for Sun Microsystems platforms developed graphical applications using APIs which tended to be tied to window systems (SunView™ and Pixrect™), hardware-specific when accelerated (GPSI and TAAClib), and based on comparatively loose and inconsistent early attempts at graphics API standards such as CORE (SunCore). None of these APIs had a means of offering a transport mechanism for truly distributed graphics-based applications.

Industry trends departed from proprietary APIs and moved toward standard APIs leading to the introduction of SunPHIGS™ and SunGKS™. However, graphic applications have also been written for a wide range of APIs that include Silicon Graphics IrisGL, Ithaca Software's HOOPS, among others, in addition to Xlib, PHIGS, and GKS. For Sun and other SPARC platforms many applications are also written to XGL, which was introduced earlier to meet the needs of Sun developers needing greater performance or more robustness in terms of features, such as immediate-mode rendering. Clearly, one company could not offer the range of APIs that were needed to cover such diverse needs.

Moreover, constant innovation in graphics techniques, such as the new multimedia-related technologies of realtime video display and virtual reality continually translates into new functionality in graphic APIs, even at the lowest layers.

IHVs have found it difficult to support third-party graphic peripherals and devices on Solaris-based platforms. Prohibitive support costs for specialized applications could lose opportunities for Solaris with OEMs. Such a result could severely limit the compatibility of Solaris with the numerous frame buffers available for x86-based platforms.

The Solaris VISUAL Strategy

To address the above problems, SunSoft has initiated a strategy for Solaris VISUAL which establishes stable, public interfaces for hardware and software developers wishing to market their product on Solaris-compatible systems.

The basic concepts of this strategy are:

- Providing in Solaris a number of *graphics foundation libraries*, covering a broad spectrum of graphics functionality.
- Enabling ISVs to provide a wide range of *graphics application libraries* on Solaris platforms via the graphics foundation libraries.
- Enabling ISVs to provide a wide range of applications on Solaris platforms via the foundation libraries, ISV-provided graphics application libraries, and a limited number of Solaris VISUAL application libraries.
- Enabling IHVs to provide a wide range of graphics hardware for Solaris-based systems and to get all Solaris-based graphics APIs and applications running on their hardware with a reasonable effort, by porting the foundation libraries.

The goal of this strategy is to provide the following balance between open systems, stability, and innovation for ISVs and IHVs:

- For ISVs, the primary architectural goals are to provide a rich set of graphics functionality and the highest performance possible through the graphics foundation libraries in a device-independent fashion. Also, this functionality is provided in a set of standard graphic APIs to provide platform independence. The programming interfaces are designed to be extensible to accommodate new innovations in graphics technology.
- For IHVs, the primary architectural goals are to provide the ability to port the Solaris VISUAL environment, including foundation libraries, standard APIs, third-party APIs, and applications, using only a device driver kit which would include a set of interface documentation, debugging harnesses, and sample code without requiring a source license. Base-level porting requires minimal effort; tuned ports, however, require some additional effort. There is also an avenue to provide IHV customers easy hardware and software installation in the field, as well as the ability to develop tools, such as test suites for IHV developers.

- Goals which SunSoft is not seeking to achieve include defining software mechanisms which require emulation of one type of hardware or another, such as interpreting statically-linked device code, defining a “Generic Frame Buffer Interface" to which device code is always expected to conform for the foundation libraries, and supporting every feature of every hardware device.

ISV and API Developer Porting

There are four graphic foundation library components:
- XIL—Imaging and digital video
- XGL— 2D/3D geometry
- PostScript—PostScript low-level rendering
- X foundation layer —Basic pixel-oriented rendering

Each of these components addresses different functionality demanded by the separate markets address. As far as possible, the components avoid redundant functions, can be used together by an application when necessary, and are not GUI- or toolkit-specific.

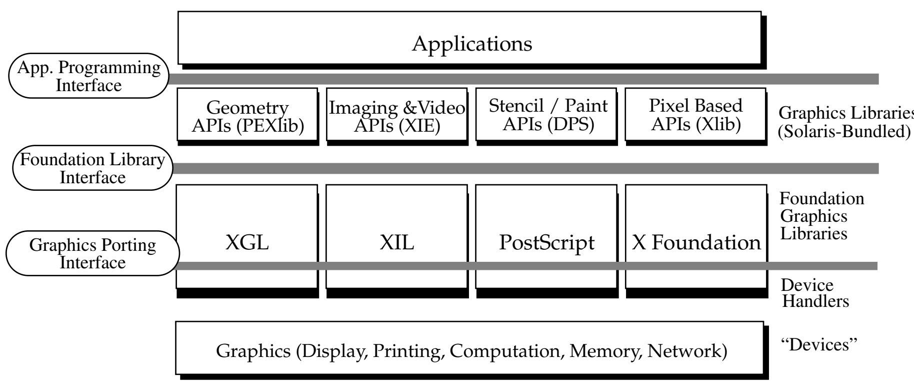

Figure 5-1 Solaris VISUAL Architecture

Architectural tenets of this architecture of concern to ISVs are:

- The window system is the centralized manager of frame buffer resources. Policies regarding the sharing of resources are implemented by the foundation library device code, under the control of the window system.

- Foundation libraries are the lowest-level device-independent programming interfaces. Some of the foundation library interfaces (for example, XGL and XIL) are available directly to ISVs for building applications or higher-level APIs. In the case of PostScript and X the standard DPSlib and Xlib interfaces are the lowest accessible libraries.
- Foundation libraries render to all frame buffers, memory rasters, and generate files suitable for hardcopy printing (including PostScript printers).
- Above the foundation library level a set of standard interfaces are offered (PEXlib, XIElib, DPSlib, and Xlib), corresponding to each one of the foundation components. XIE is still under development by the MIT consortium and is not yet available under Solaris.

Solaris VISUAL libraries make an important commitment to remote graphics along the X model, which is that a client program can be remote from the machine driving the display. Any graphics programming interface available to ISVs will operate to a remote X-server through the X protocol or one of its extensions—PEX, XIE, or DPS.

Note that a common characteristic of the graphics programming interfaces available to ISVs is that they rely on the application program for most or all of the windowing functionality, such as creating windows, managing input events, and creating an X drawable through X and then using one of the Solaris VISUAL APIs, such as DPS or XIL to render into the created X drawable. A notable exception is Xlib which, in addition to functioning as a graphics application library, also functions as the base level for window toolkit APIs, and, therefore, must provide windowing functionality as well.

Within these foundation libraries you can also use *Direct Graphics Access* (DGA). DGA allows a graphics application process to access graphics devices directly and to bypass the X protocol when the application runs on the same machine as the window server. DGA manages this access along with the X-server so that multiple clients do not collide. Libraries such as XGL, XIL, and Direct Xlib (DGA-based Xlib) use this feature to obtain higher performance for many graphic operations on a local host.

Application Graphics Libraries are APIs that are layered on top of one or more of the foundation graphics libraries. Solaris VISUAL includes four application graphics APIs that correspond to each of the following foundation graphics libraries (API/wire protocol/foundation library):

- Xlib/X11/X Foundation—Standard MIT Xlib library API.
- XIElib/XIE/XIL—API to MIT’s X Imaging Extension (XIE). This is still under development by MIT.
- DPSlib/DPS/PostScript—Display PostScript API from Adobe.
- PEXlib/PEX/XGL—API for accessing MIT’s 3D graphics PEX extension.

To achieve portability in your applications across multiple OS platforms, build your applications to one or more of the Solaris application library interfaces: PEXlib, DPS, XIE, or Xlib. Another alternative is to use one of several third-party multi-platform APIs, such as Ithaca Software HOOPS, the Nth graphic NPGL (IrisGL library), or PHIGS. If you are concerned about performance or gaining functionality not provided by any of these layered APIs, you can directly program to XIL or XGL.

Programmers are strongly encouraged to build their applications using dynamic linking. See the Chapter 1 on "Dynamic Linking" in this book. Over time, support for static linking to any Solaris VISUAL library is likely to be removed.

Note – With the current release of OpenWindows, PostScript rendering is achieved through NeWS and tNt. Keep in mind that in future releases of Solaris, PostScript rendering will be accomplished through Display PostScript (DPS).

Note – When moving to Solaris 2.x, Sun Microsystems indicated its intent to phase out support for Pixrect/Pixwins and SunView applications running alongside OpenWindows. Until Solaris 2.x, Sun Microsystems supported building and running applications for SunView, as well as running them under an emulator in OpenWindows. Starting with Solaris 2.x, libraries for building SunView and Pixrect/Pixwins-based tools and applications are no longer present. Only SunView binaries dynamically linked under 4.1.x running in binary compatibility mode are supported under Solaris 2.x; over time, support for this will disappear as well. Therefore, ISVs who still have SunView or Pixrect/Pixwins code in their applications must port them to a supported OpenWindows-based API to port them to Solaris 2.x.

IHV Porting

For a display device vendor, porting the Solaris VISUAL environment to a new graphics device involves the following:

- Writing device handlers to support one or more Graphics Porting Interfaces (GPI) which are defined by the different graphics foundation libraries.
- Porting a single GPI handler interface for the X foundation layer (the MIT standard DDX layer). Given that all other foundation libraries can run on the X foundation layer via a default device handler, as indicated in Figure 5-2, this interface provides a quick solution to support all APIs, albeit with performance degradation in some cases.
- Writing additional handlers that allow greater optimization and use extended hardware features of the target device that function at a higher level in a given foundation API than the base X foundation layer allows.

- Providing a loadable kernel device driver for graphic devices. Generally, these drivers do not perform graphics operations; they initialize the device, perform policy-free allocation of device resources, map the device into end-user process address spaces, update control registers on the device which are unsafe to map into end-user address spaces, perform time-critical device update functions, such as those which must synchronize with vertical retrace, track hardware cursors, and context-switch the graphics device. The SunSoft DDI/DKI will provide support for writing such drivers and sample code will be available.

Figure 5-2 shows the independent hardware vendor (IHV) view of foundation library interdependencies and porting opportunities, with device handlers in white. The details of these device handlers are described in further detail later in this chapter.

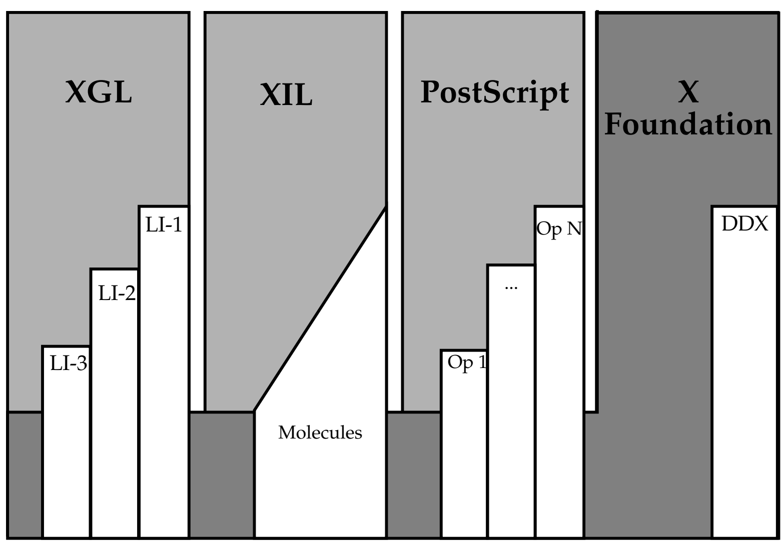

Figure 5-2 Foundation Library Interdependencies and Porting Opportunities

Graphics Software Interfaces

The following section discusses the foundation libraries, as well as some of the libraries bundled with Solaris 2.x.

XGL Overview

XGL is an immediate-mode foundation graphics library which provides the geometry-oriented graphics component of Solaris VISUAL from SunSoft. It functions in the OpenWindows environment and provides high functionality and high performance for both 2D and 3D graphics. XGL utilizes the SunSoft Direct Graphics Access (DGA) technology. DGA provides high performance by coordinating access directly to graphics accelerators such as Sun Microsystems Computer Corporation's GX, GS, and GT accelerators and, in the future, other SPARC and x86 system graphics accelerators supported under the Solaris 2.x environment.

XGL features an object-oriented style programming interface benefiting from the classing architecture of C++. The display primitives are similar to PHIGS and include arcs, circles, polylines, polygons, triangle strips, quadrilateral meshes, and, in XGL 3.0.x, NURBS curves and surfaces.

The following table shows the latest version of XGL 3.0.1:

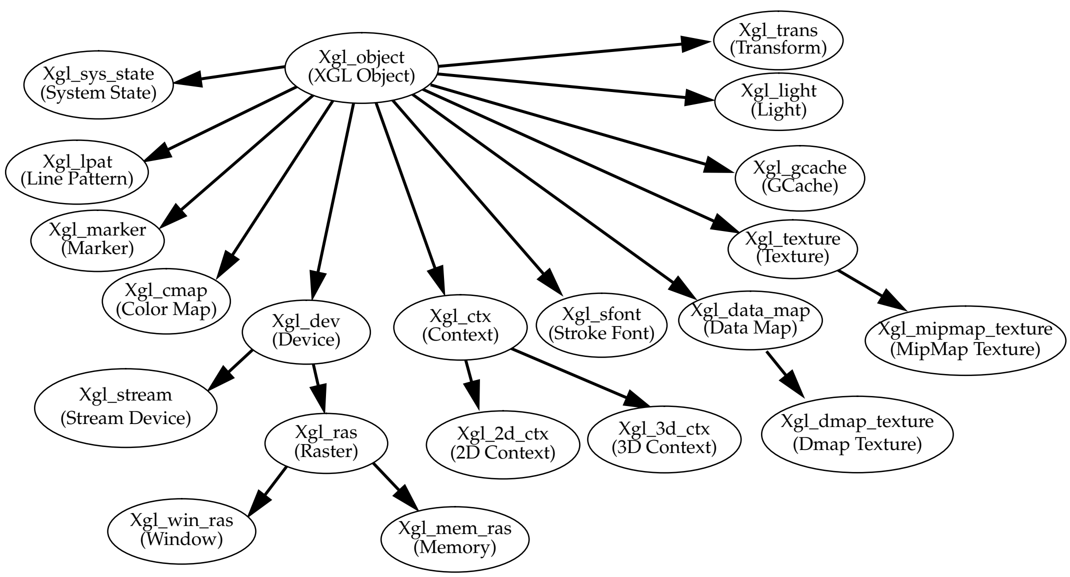

Figure 5-3 Complete XGL 3.0.1 API Class Hierarchy

Because XGL is an immediate mode library, it gives applications direct control of screen graphics. XGL is well suited for existing applications that maintain their own display lists and are difficult to fit into the PHIGS or GKS programming models. It is also appropriate for applications that need the highest possible performance from graphics accelerators.

XGL provides compatible, separate 2D and 3D graphics pipelines. Separate pipelines allow 2D rendering to achieve greater performance without the overhead of 3D computation.

A unique XGL feature is multi-primitive calls. This feature permits batching of primitives into groups to minimize procedure-call overhead. Batching can provide substantial performance gains for applications that repeatedly render the same type of primitive, such as the rectangles and circles used by electronic design.

XGL 3.0.1 Features

XGL version 3.0.1 is supported under Solaris 2.x (see Figure 5-3). In addition, it extends functionality from XGL 2.0 to incorporate new functionality, described in Table 5-1.

Table 5-1 XGL 3.0.x New Features

API or LI Definition	SW Pipeline	New Feature
X (new to 3.0.1)	X (new to 3.0.1)	Support for stream devices
X (new to 3.0.1)	NA	Support for hardware Gcaches
X	X	Loadable pipelines
X	X	New technology for NURBS curves
X	X	NURBS surfaces with dynamic tessellation
X	X (new to 3.0.1)	Support for texture mapping
X	X (new to 3.0.1)	Antia-liasing
X	X	General support for backing store
X	X	Xlib true double buffering
X	X (new to 3.0.1)	Transparency attributes
X	X (new to 3.0.1)	Accumulation buffer
X	NA	Support for stereo-capable devices
X	X	Control of Zbuffer test
X	X	Triangle-list primitive
X	X	Support for double precision
X	X	Sub-pixel accurate 3D rendering
X	X	Optimization flags for transform objects
X	X	Better support for bounding-box checking
X	X	User accessible MC to DC transform
X	X	New optimization flags for simple polygons
X	NA	Better looking stroke fonts
X	NA	ANSI C compliance
X	X	PEX support

Obsolete XGL 2.0 features include fract point types, 3D integer point types, and individual object creation functions.

Also incorporated in the XGL ISV (Application Developer's) Kit is a tool that will help highlight code the must be changed when porting programs from XGL 2.0 to XGL 3.0.x.

Features introduced into XGL 3.0.1 and not in XGL 3.0 include a device object to support stream output. This capability will aid SunSoft as well as third-party vendors to develop pipelines for supporting CGM metafiles and PostScript. Also, XGL can use a hardware geometry cache (`Gcache`), if one is present in the graphics hardware, rather than providing software emulation.

Also introduced in XGL 3.0.1 are new texture mapping features. Two new objects are used to support 2D texturing of surface primitives:

- `Xgl_mipmap_texture`, the object containing the textured image
- `Xgl_dmap_texture`, the object that specifies the control parameters needed for texturing when rendering through a 3D context

XGL uses the texturing image specified by the MipMap Texture object and maps the texture onto a polygon face by listing *u* and *v* values in the "data" fields of the vertices. XGL 3.0.1 supports texturing to be applied at different stages in the rendering pipeline. It also supports different color composition capabilities such as blend, decal, and modulate. The MipMap Texture object(s) to use and the interpolation method and color composition method to use are encapsulated in the Dmap Texture object. Each 3D context can also have multiple Dmap Texture objects.

XGL Loadable Pipelines

Loadable interfaces (pipelines) are a powerful new feature of XGL 3.0.x with the XGL IHV (Hardware Developer's) Kit, which will be available in the fall of 1993. There are three layers of rendering pipelines: LI-1, LI-2, and LI-3.

Loadable Interface 3 (LI-3) is the lowest layer in the rendering pipeline and is the closest to the X foundation graphics library that supports the lowest layer for XIL and PostScript rendering. This layer is responsible for rendering pixels individually or in spans. If you are porting a device pipeline to this layer, you only need to implement span and dot renderers. All other operations needed to process an API primitive and reduce it to this level are provided by XGL's default software implementation included in XGL 3.0. All new devices supporting XGL must minimally implement this layer, although XGL does supply example LI-3 layers for memory devices that can be used to assist porting efforts.

Loadable Interface 2 (LI-2) is the second layer, primarily responsible for scan-converting more complex primitives like polygons and polylines. This layer assumes more responsibility for rendering, but does not yet process geometry transformations and clipping.

Loadable Interface 1 (LI-1) is the layer directly below the API. This layer is the most difficult layer to port to, but it is most useful for developers wishing to maximize performance through device-dependent support of XGL on devices that directly support lighting, depth-cueing, and transformations.

By offering these three layers of software, an IHV can enable its customers to support either dumb frame buffers such as SPARC's CG3 or an SVGA card on an x86 system using the LI-3 layer, or something more complex, such as an optimized port on a GT board.

Because the hardware pipelines are dynamically loaded, ISVs and end users can build applications to XGL 3.0.x without concern about having multiple versions built for different display hardware.

Hardware pipelines also introduce a new way of architecting and packaging XGL, enabling third-party vendors to plug in support for their boards on their own CD ROMs without having to be a part of the XGL core distribution. Figure 5-4 illustrates packaging models.

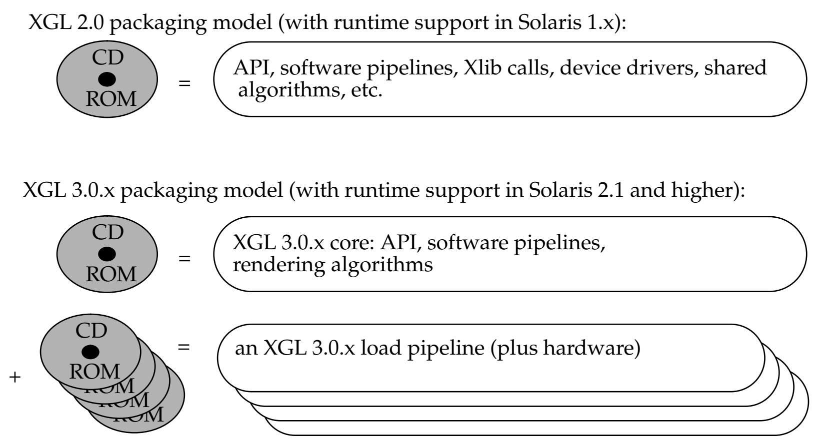

Figure 5-4 XGL Software Packaging

Another unique feature of XGL 3.0.x is that it lets you develop drivers simultaneously to three different layers of pipelines. The number of layers you port to will depend on whether the hardware manufacturer can provide various stages of acceleration in hardware to take advantage of supporting a given layer.

XGL 3.0.x is packaged in the following manner:

Table 5-2 XGL 3.0.x Product Packaging

Product	Solaris 2.1 and higher
XGL RTE (Runtime Environment)	Bundled
XGL ISV (Application Developer's) Kit: to develop applications	Unbundled
XGL IHV (Hardware Developer's) Kit: for hardware pipelines	Unbundled[1]

1. Available in the fall of 1993.

An OLIT Program Using XGL

The following example program `ow_olit.c` uses the OLIT toolkit to create the window used by XGL 3.0.x:

Code Example 5-1 `ow_olit.c`

```
/* ow_olit.c - attach XGL Window Raster to OLIT stub widget */

/*
 * This program demonstrates how to create an XGL Window Raster given
 * an OLIT stub widget.
 */

#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xol/OpenLook.h>
#include <Xol/OblongButt.h>
#include <Xol/DrawArea.h>

#include <xgl/xgl.h>

/*
 * need these global so that expose procedure can get to it
 */
static Xgl_2d_ctxctx = NULL; /* XGL Context object */
static Xgl_sys_statesys_st; /* XGL System State Object */

main (argc, argv)
 int    argc;
 char*argv[];
{
 Xgl_win_ras win_ras = NULL;/* Xgl window raster */
 Xgl_X_window xgl_x_win;/* XGL-X data structure */
 Xgl_obj_desc win_desc;/* XGL window raster struct */
```

Code Example 5-1 `ow_olit.c` *(Continued)*

```
Widget toplevel;/* Widget top-level */
Widget drawable;/* Drawable widget */

Window window;/* X window for canvas */
Display *display;/* pointer to X display */

void repaint_proc ();/* OLIT repaint procedure */

/*
* Initialize toolkit and make frame
*/
toplevel = OlInitialize (argv[0], "Xgl-Widget", NULL, 0, &argc,
          argv);

/*
* Create a drawable canvas - i.e., create a widget which can
* be used for rendering xlib graphics
*/
drawable = XtVaCreateManagedWidget( "Simple OLIT Drawable",
          drawAreaWidgetClass, toplevel,
          XtVaTypedArg, XtNvisual, XtRString,
          "PseudoColor", sizeof("PseudoColor"),
          XtNlayout, OL_IGNORE,
          XtNheight, 512,
          XtNwidth, 512,
          NULL);

/*
* Add Expose Callback to Draw Area Widget
*/
XtAddCallback(drawable, XtNexposeCallback, repaint_proc,
       (XtPointer) NULL);
XtRealizeWidget(toplevel);

/* get X stuff */
display = (Display *) XtDisplay (drawable);
window = (Window) XtWindow (drawable);

/* put X stuff into XGL data structure */
xgl_x_win.X_display = (void *) display;
xgl_x_win.X_window = (Xgl_usgn32) window;
xgl_x_win.X_screen = (int) DefaultScreen(display);

/* wait for the window */
```

Code Example 5-1 `ow_olit.c` *(Continued)*

```
  sleep (2);

  /* create XGL System State object */
  sys_st = xgl_open(NULL);

  /* create Window Raster Device using OLIT widget */
  win_desc.win_ras.type = XGL_WIN_X;
  win_desc.win_ras.desc = &xgl_x_win;
  win_ras = xgl_object_create (sys_st, XGL_WIN_RAS, &win_desc,
             NULL);

  /* create XGL graphics Context object using the Window Raster
  object */
  ctx = xgl_object_create (sys_st, XGL_2D_CTX, NULL,
             XGL_CTX_DEVICE, win_ras,
             NULL);

  /* go into Xt main loop and wait for expose events */
  XtMainLoop();

  Exit (0);
 }

/*
 * Xgl repaint procedure called on expose events
 */
static
void
repaint_proc ()
{
  Xgl_pt_listpl;/* polygon point structure */
  Xgl_pt_i2dpts_i2d[3];/* integer 2d points */

  /* clear canvas area to default background color of black */
  xgl_context_new_frame (ctx);

  /* setup data points for a triangle - XGL will close triangle */
  pts_i2d[0].x = 25;
  pts_i2d[0].y = 50;
  pts_i2d[1].x = 50;
  pts_i2d[1].y = 400;
  pts_i2d[2].x = 250;
  pts_i2d[2].y = 375;

  /* fill XGL polygon data structure */
```

Code Example 5-1 `ow_olit.c` *(Continued)*

```
 pl.pt_type = XGL_PT_I2D;
 pl.num_pts = 3;
 pl.bbox = 0;
 pl.pts.i2d = &pts_i2d[0];

 /* draw a polygon in the default color of white */
 xgl_polygon (ctx, XGL_FACET_NONE, 0, 0, 1, &pl);
xgl_context_flush(ctx, XGL_FLUSH_BUFFERS);
}
```

XIL Overview

The Solaris XIL Imaging Library provides a basic set of functions for imaging and digital video applications. The XIL software contains three primary components:

- A C programming interface specification for basic imaging and digital video functionality.
- A high-performance implementation of the specification tuned for specific Solaris platforms.
- A standard hardware interface specification (C++), which enables third-party developers to readily support the XIL library and applications built on the XIL library. The hooks that IHV developers develop are referred to as *molecules* which define groups of atomic XIL functionality that can be combined together and executed more efficiently on the IHV's target device.

Note – Developers should be aware that as a new API for Sun, XIL will be available only on Solaris 2.x and future versions of Solaris; if you want to utilize XIL functionality in your applications, plan your Solaris 2.x migration efforts accordingly. The XIL 1.0 library runs on Sun-4c and Sun-4m workstations or on servers in the OpenWindows Version 3.1 environment.

In general XIL as a rendering library coexists with and complements the X windowing system rather than superseding or duplicating its functionality, as follows:

- Like XGL and DPS, you use X to create a window and an X drawable and then subsequently render into this drawable with XIL calls.
- You create a color map through X and then use XIL functions to make an image fit that color map. XIL itself does not manipulate color maps.

- XIL does not display images to match the display depth of the X server's visual. For example, it will not implicitly render a 24-bit image on an 8-bit visual. Applications must use XIL functions to explicitly convert an image to the appropriate type that the X-server expects before displaying it. In this case, it is the actual converting of imagery to different display depths where XIL adds its value.
- XIL does not provide any backing store. Performance-driven applications must keep a retained image through X if fast redraws are desired.

An important feature of XIL is its ability to use implicit pipelining and deferred execution within the API. Doing so simplifies the API, allows code optimization, and improves XIL performance

Without pipelining:

```
temp=A*B; D=temp+C
```

With pipelining:

```
D=A*B+C
```

Typically, pipelining is identified in source code explicitly, as in this example:

```
begin_pipeline() ;
add(a,b,c) ;
multiply(c,d,e) ;
end_pipeline() ;
execute_pipeline() ;
```

XIL avoids the need to bracket its calls with pipelining statements. Instead, it recognizes which operations can be pipelined automatically and does the pipelining implicitly, as in this example:

```
xil_add(a,b,c) ;
xil_multiply(c,d,e) ;
```

By implicitly pipelining instructions, XIL simplifies the API and provides opportunities to optimize the code. By deferring execution of XIL instructions transparently, XIL performance is improved in the following ways:

- Deferred operations are processed all at once.
- Chains of operations are recognized and performed with special optimized functions.
- The chains that are recognized are specified by device drivers when they are loaded.

The XIL library is complemented by the XGL Graphics Library, which addresses application and API requirements for geometry-based graphics. Functions from the two libraries can be called from within the same program, and XIL supplies functions to convert between XIL images and XGL memory rasters, or between XIL display images and XGL window rasters. As with XGL, XIL-based applications use X library calls to create and draw into windows and to create and manipulate color maps.

XIL Objects

Like XGL, XIL is an object-oriented interface. The details of its implementation are hidden from the programmer, enabling SunSoft to extend XIL over time without changing the interface. Figure 5-5 shows the hierarchy of XIL data objects which consist of the state information that XIL operators need:

.

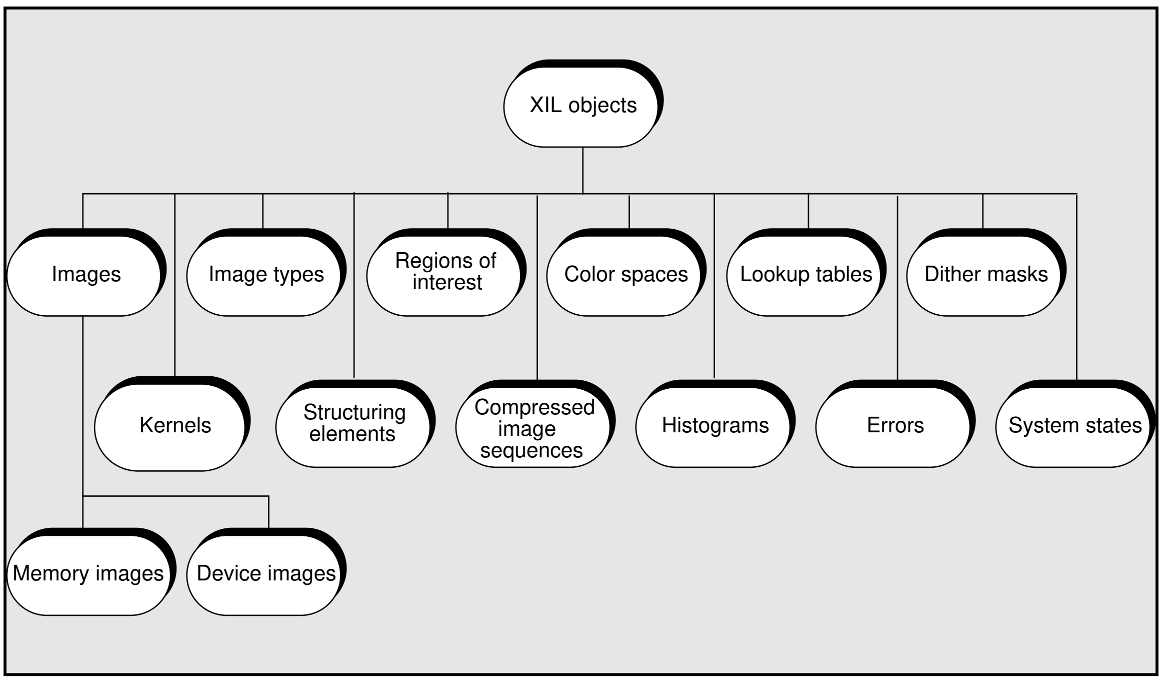

Figure 5-5 XIL Data Objects

The following describes each of the objects in Figure 5-5:

- **The Image Object**
  The most important XIL data object is the image. This object stores a set of pixel values and values for a number of image attributes, such as width, height, number of bands, and data type. The library supports images of virtually any size: an image's width, height, and number of bands can each be as great as 2^{32}-1. In addition, the library supports three types of data: 1-bit data, 8-bit unsigned data, and 16-bit signed data.

  The XIL library does not currently support multiband images in which different bands have different dimensions. Thus, there is no direct support of 4:1:1 or 4:2:2 data.

- **Image Types**
  For every XIL image, there is a subset of attributes that constitutes its image type. These attributes are its width, height, number of bands, data type, and color space. An application might get an image's image type in preparation for creating an X window in which to display the image.

- **Regions of Interest**
  In the XIL scheme, a region of interest is attached to an image rather than to an operation. This region of interest is a single-bit mask that determines either which pixels in a source image can be read or which pixels in a destination image can be written.

- **Color Spaces**
  A color space data object stores information about an image's color space. The color spaces supported by the XIL library include:

  - CCIR Rec. 709 RGB
  - A linear version of CCIR Rec. 709 RGB
  - CCIR Rec. 709 YCC
  - A luminance-only space derived from CCIR Rec. 709 YCC
  - A linear version of the luminance-only space mentioned above
  - CCIR Rec. 601 YCC
  - A luminance-only space derived from CCIR Rec. 601 YCC
  - A YCC space defined by Eastman Kodak Company for its Photo CD™ product
  - A linear CMY
  - A linear CMYK

  An application generally attaches color space objects to images in preparation for converting an image from one color space to another.

- **Lookup Tables**
  Lookup tables are used in two ways in XIL applications. First, they are used for general image modification. A single-band image of any data type can be converted to a single-band or multiband image of any data type. Second, lookup tables are used in dithering operations.

- **Dither Masks**
  A dither-mask object is one of the required parameters to the XIL library's ordered-dither operator. The mask is a three-dimensional array whose depth matches the number of bands in the image being dithered. The values in the mask act as threshold values that help eliminate contouring in the dithered image.
- **Kernels**
  A kernel is one of the required parameters to the convolution operator. The values in the kernel determine the effect that the convolution operation will have. For example, the operation may sharpen an image, smooth an image, or highlight the edges in an image.
- **Structuring Elements**
  A structuring element serves as a parameter to the erosion and dilation operators and defines the neighborhood of pixels in the source image to be used in calculating the value of each pixel in the destination image.
- **Compressed Image Sequences**
  A compressed image sequence is a container for compressed images. These images may represent a series of still images, frames from a movie, or pages from a document, and they may have been compressed in one of several formats: Cell (a format developed at Sun), JPEG baseline sequential or lossless, and CCITT Group 3 or Group 4.
- **Histograms**
  When an XIL application calculates an image's histogram, it writes its results to a histogram object, containing a set of bins that stores information about gray or color levels in the image.
- **Errors**
  When an XIL error occurs, the library generates an error object. This object contains such information as an error ID and a string containing a description of the error. The XIL library includes a default error handler that gets values from this object and prints a message to `stderr`. It also contains a set of error handling functions that get information from this type of object. You can use these functions to construct a custom error handler.
- **System States**
  A system-state object is created when an application initializes the library. Among other things, this object is used to maintain a list of the data objects the application has created and to keep track of the current error handler.

XIL Functions

XIL incorporates a suite of image processing, digital video, color, preparing images for output, and error handling. Image processing functions range from basic imaging operations commonly found on many frame buffers to operations required to support more sophisticated imaging APIs. APIs that wish to provide a higher level of functionality for more specific areas can make XIL calls and call the XIL export function to touch image data directly. All imaging access to the frame buffer and hardware accelerators is through XIL.

The following tables show the set of functions available for XIL 1.0:

Table 5-3 Arithmetic and Logical Functions Involving Two Source Images

Function Name	Description
`xil_add`	Adds two images
`xil_subtract`	Subtracts one image from another
`xil_multiply`	Multiplies two images
`xil_divide`	Divides one image into another
`xil_and`	Finds the logical AND of two images
`xil_or`	Finds the logical OR of two images
`xil_xor`	Finds the exclusive OR of two images

Table 5-4 Arithmetic and Logical Functions Involving a Source Image and a Constant

Function Name	Description
`xil_add_const`	Adds an image and a constant
`xil_subtract_const`	Subtracts a constant from an image
`xil_subtract_from_const`	Subtracts an image from a constant
`xil_multiply_const`	Multiplies an image by a constant
`xil_divide_by_const`	Divides an image by a constant
`xil_divide_into_const`	Divides an image into a constant
`xil_and_const`	Finds the logical AND of an image and a constant
`xil_or_const`	Finds the logical OR of an image and a constant
`xil_xor_const`	Finds the exclusive OR of an image and a constant

Table 5-5 Geometric Operations

Function Name	Description
xil_translate	Moves an image up or down and/or left or right
xil_scale	Increases or decreases the width and/or height of an image.
xil_subsample_adaptive	Reduces the size of an image using an algorithm that takes into consideration the value of each pixel in the source image.
xil_subsample_binary_to_gray	Scales down a 1-bit image, making up for a loss in spatial resolution with an increase in grayscale resolution.
xil_rotate	Rotates an image
xil_affine	Performs affine transforms, including scaling, rotation, translation, and shearing.
xil_transpose	Reflects an image in one of several directions, or rotates it in multiples of 90 degrees.

Table 5-6 Miscellaneous Image Processing Operations

Function Name	Description
xil_extrema	Finds the minimum and maximum values in an image
xil_histogram	Produces a histogram for an image
xil_threshold	Thresholds an image
xil_fill	Performs a boundary fill
xil_soft_fill	Performs a soft fill
xil_convolve	Filters an image
xil_dilate	Dilates an image
xil_erode	Erodes an image
xil_lookup	Passes an image through a look-up table
xil_blend	Blends an image
xil_paint	Paints on an image

Table 5-6 Miscellaneous Image Processing Operations (Continued)

Function Name	Description
xil_set_pixel	Sets the value of a single pixel
xil_get_pixel	Reads the value of a single pixel
xil_set_value	Sets all the pixels in an image to a constant
xil_copy_pattern	Copies a pattern to an image

Table 5-7 Image Compression Functions

Function Name	Description
xil_compress	Compress an XIL image and writes the compressed data to a compressed image sequence (CIS)
xil_cis_put_bits	Copies compressed data into a CIS
xil_cis_put_bits_ptr	Give a CIS a pointer to a compressed datastream
xil_decompress	Decompresses a compressed image
xil_cis_get_bits_ptr	Retrieves a pointer to a CIS's compressed data
xil_cis_has_frame	Determines whether a CIS contains at least one frame of compressed data
xil_cis_has_data	Returns the number of unread bytes in a CIS
xil_cis_number_of_frames	Returns the number of unread frames in a CIS
xil_cis_seek	Seeks to a particular frame of compressed data in a CIS
xil_cis_reset	Clears all existing data from a CIS
xil_cis_attempt_recovery	Attempts to recover from a datastream error

Table 5-8 Presentation Functions

Function Name	Description
xil_copy	Copies an image (to a display or printer)
xil_rescale	Maps the values in each band of an image from one range to another
xil_cast	Casts an image from one data type to another
xil_nearest_color	Dithers an image by finding the nearest match between colors in the source image and colors in a look-up table

Table 5-8 Presentation Functions (Continued)

Function Name	Description
xil_error_diffusion	Performs error diffusion dithering
xil_ordered_dither	Performs an ordered dither
xil_color_convert	Converts an image from one color space to another
xil_black_generation	Performs undercolor removal and black generation on CMYK images

Table 5-9 Error Handling Functions

Function Name	Description
xil_error_get_string	Returns a string that describes the error that just occurred
xil_error_get_id	Returns the error ID of the last error to occur
xil_error_get_category	Returns a constant indicating the type of error that occurred. Possible error types include resource errors, arithmetic errors, and user errors, etc.
xil_error_get_category_string	Returns a string indicating the type of error that occurred
xil_error_get_primary	Indicates whether an error is the primary cause of a problem you are seeing (or a secondary error)
xil_error_get_location	Returns encrypted information about where in the XIL library an error was generated
xil_error_get_object	Returns a handle to the XIL data object affected by the error

For more information on XIL consult the *Solaris XIL Imaging Reference Manual*, which contains man pages for all the functions in the XIL library, and the *Solaris XIL 1.0 Imaging Library Programmer's Guide*.

An XIL Program

The following is an example of an XIL-based program which reads in a grayscale image from a file and displays that image in an Xlib window. The example is divided into three segments: `example1.c`, `fileio.c`, and `windows.c`:

Code Example 5-2 example1.c

```
/* This example reads a grayscale image from a file and displays the image
 in an X window
*/
#include <stdio.h>
#include <stdlib.h>
#include <xil/xil.h>

#define NCOLORS 216

XilImage load_file();

Display *xdisplay;
Window xwindow;

main(int argc, char **argv)
{
    char filename[256];
    XilSystemState state;
    XilImage image, bit_image, retained_image, display, band0, band1, band2;
    unsigned int width, height, nbands;
    XilDataType datatype;
    XVisualInfo vinfo;
    XEvent event;
    float mult[1], offset[1];
    XilLookup color map;
    XilKernel distribution;

    /* multiplier == -1 and offset == 0 assumes color map[0] is white,
     * color map[1] is black */
    static int multipliers[1] = {-1};
    static unsigned int dimensions[1] = {2};

    /* Get filename from command line */
    *filename = NULL;
    while (--argc) {
       if ((*++argv)[0] == '-')
           switch ((*argv)[1]) {
           case 'h':
               fprintf(stderr, "Usage: example1 [header filename]\n");
```

Code Example 5-2 example1.c (Continued)

```
                break;
            }
        else {
            strcpy(filename, *argv);
        }
    }
    if (*filename == NULL) {
        fprintf(stderr, "Usage: example1 [header filename]\n");
        exit(1);
    }

    state = xil_open();
    if (state == NULL) {
        /* XIL's default error handler will print an error msg on stderr */
        exit(1);
    }
    image = load_file(state, filename);
    xil_get_info(image, &width, &height, &nbands, &datatype);

    if ((datatype != XIL_BYTE) || (nbands != 1)) {
        fprintf(stderr, "This program requires a single band, 8 bit image\n");
        exit(1);
    }

    if (open_window(width, height, &vinfo) == 0) {
        exit(1);
    }

    /* Wait for the window to be mapped (an Expose event) */
    while (1) {
        XNextEvent(xdisplay, &event);
        if (event.xany.type == Expose)
            break;
    }

    /* Create a display image */
    display = xil_create_from_window(state, xdisplay, xwindow);

    /* Make changes to source image so that it can be copied to the display */
    if (vinfo.class == TrueColor) {
        /* Copy the grayscale image into each of the bands */
        retained_image = xil_create(state, width, height, 3, XIL_BYTE);
        band0 = xil_create_child(retained_image, 0, 0, width, height, 0, 1);
        band1 = xil_create_child(retained_image, 0, 0, width, height, 1, 1);
        band2 = xil_create_child(retained_image, 0, 0, width, height, 2, 1);
```

Code Example 5-2 example1.c (Continued)

```
        xil_copy(image, band0);
        xil_copy(image, band1);
        xil_copy(image, band2);
    }
    else if (vinfo.class == GrayScale) {
        /* Remap the image to fill NCOLORS entries */
        mult[0] = NCOLORS / 256.0;
        offset[0] = 256 - NCOLORS;
        xil_rescale(image, image, mult, offset);
        retained_image = image;

    }
    else if ((vinfo.class == StaticGray) && (vinfo.depth == 8)) {
        retained_image = image;
    }
    else if ((vinfo.class == StaticGray) && (vinfo.depth == 1)) {
        bit_image = xil_create(state, width, height, 1, XIL_BIT);
        if (bit_image == NULL) {
            /* XIL default error handler will print an error msg to stderr */
            exit(1);
        }
        distribution = xil_kernel_get_by_name(state, "floyd-steinberg");
        color map = xil_colorcube_create(state, XIL_BIT, XIL_BYTE, 1, 0,
                                    multipliers, dimensions);
        xil_error_diffusion(image, bit_image, color map, distribution);
        retained_image = bit_image;
    }

    xil_copy(retained_image, display);

    /* Refresh window as necessary till program terminates */
    while (1) {
        XNextEvent(xdisplay, &event);
        if (event.xany.type == KeyPress) {
            xil_not(retained_image, retained_image);
            if (vinfo.class == GrayScale)
                xil_add_const(retained_image, offset, retained_image);
            xil_copy(retained_image, display);
        }
        else if (event.xany.type == Expose)
            xil_copy(retained_image, display);
        else if (event.xany.type == ButtonPress)
            break;
    }
```

Code Example 5-2 example1.c (Continued)

```
    /* Cleanup */
    XCloseDisplay(xdisplay);
    xil_close(state);
}
```

Code Example 5-3 fileio.c

```
#include <stdio.h>
#include <stdlib.h>
#include <xil/xil.h>

XilImage
load_file(XilSystemState state, char *filename)
{
    XilImage image;
    unsigned int width, height, nbands, bits;
    XilDataType datatype;
    XilMemoryStorage storage;
    Xil_boolean status;
    FILE *in_file;
    char datafile[256];
    int size;

    /* Open the header file */
    if ((in_file = fopen(filename, "r")) == NULL) {
        fprintf(stderr, "Could not open %s\n", filename);
        exit(1);
    }

    /* Get header information */
    fscanf(in_file, "%d%d%d%d", &width, &height, &bits, &nbands);
    fscanf(in_file, "%s", datafile);
    fclose(in_file);

    if ((bits == 8) && (nbands == 1)) {
        datatype = XIL_BYTE;
    }
    else {
        fprintf(stderr, "This program requires a single band, 8 bit image\n");
        exit(1);
    }

    /* Open the data file */
```

Code Example 5-3 fileio.c (Continued)

```
    if ((in_file = fopen(datafile, “r”)) == NULL) {
        fprintf(stderr, “Could not open %s\n”, datafile);
        exit(1);
    }

    /* Create the image to read the data into */
    image = xil_create(state, width, height, nbands, datatype);
    if (image == NULL) {
        /* XIL’s default error handler will print an error msg to stderr */
        exit(1);
    }

    /* Get the pointer to the image data */
    if (xil_export(image) == XIL_FAILURE) {
        /* XIL’s default error handler will print an error msg to stderr */
        exit(1);
    }
    status = xil_get_memory_storage(image, &storage);
    if (status == FALSE) {
        /* XIL’s default error handler will print an error msg to stderr */
        exit(1);
    }

    /* Load the image data */
    size = nbands * width * height;
    if (fread((char *) storage.byte.data, sizeof(Xil_unsigned8),
              size, in_file) != size) {
        fprintf(stderr, “Error reading data in %s\n”, datafile);
        exit(1);
    }
    fclose(in_file);

    /* Make the image available for XIL processing */
    xil_import(image, 1);
    return (image);
}
```

Code Example 5-4 window.c

```
#define MAXCOLORS 256

#include <stdio.h>
#include <stdlib.h>
```

Code Example 5-4 window.c (Continued)

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

#define NCOLORS 216

extern Display *xdisplay;
extern Window xwindow;

int errhandler(Display *);

int
open_window(width, height, vinfo)
    u_int width, height;
    XVisualInfo *vinfo;
{
    XSetWindowAttributes setwinattr;
    color map color map;
    unsigned long pixels[MAXCOLORS], pmask[1], pixel[1];
    int i, j, k, index;
    XColor colors[MAXCOLORS];
    int base;

    xdisplay = XOpenDisplay(NULL);
    if (xdisplay == NULL) {
       fprintf(stderr, "Unable to connect to server\n");
       return (0);
    }

    XSetIOErrorHandler(errhandler);

    /* Find best visual */
    if (XMatchVisualInfo(xdisplay, DefaultScreen(xdisplay), 24,
                       TrueColor, vinfo) &&
            (DefaultDepth(xdisplay, DefaultScreen(xdisplay)) == 24)) {
    }
    else if (XMatchVisualInfo(xdisplay, DefaultScreen(xdisplay), 8,
                            GrayScale, vinfo)) {
    }
    else if (XMatchVisualInfo(xdisplay, DefaultScreen(xdisplay), 8,
                            StaticGray, vinfo)) {
    }
    else if (XMatchVisualInfo(xdisplay, DefaultScreen(xdisplay), 1,
                            StaticGray, vinfo)) {
    }
    else {
       return (0);
```

Code Example 5-4 window.c (Continued)

```
    }

    color map = XCreatecolor map(xdisplay, DefaultRootWindow(xdisplay),
                                vinfo->visual, AllocNone);
    setwinattr.color map = color map;

    xwindow = XCreateWindow(xdisplay, DefaultRootWindow(xdisplay),
         20, 20, width, height, 0, vinfo->depth, InputOutput, vinfo->visual,
                          CWcolor map, &setwinattr);

    if (xwindow == NULL) {
        fprintf(stderr, "Unable to create window\n");
        return (0);
    }
    if (vinfo->class == GrayScale) {
        base = MAXCOLORS - NCOLORS;

        i = XAllocColorCells(xdisplay, color map, 1, pmask, 0, pixels, 256);

        for (pixel[0] = 0; pixel[0] < base; pixel[0]++) {
            XFreeColors(xdisplay, color map, pixel, 1, 0);
        }

        /* Fill the color map */
        index = 0;
        for (i = 0; i < NCOLORS; i++) {
            colors[index].pixel = index + base;
            colors[index].blue = colors[index].green = colors[index].red =
                (255 * i / (NCOLORS - 1)) * 256;
            colors[index].flags = DoRed | DoGreen | DoBlue;
            index++;
        }
        XStoreColors(xdisplay, color map, colors, NCOLORS);
    }
    /* List window events used */
    XSelectInput(xdisplay, xwindow, ExposureMask | ButtonPressMask |
                KeyPressMask);

    /* Make the window visible */
    XMapWindow(xdisplay, xwindow);
    return (1);
}

int
errhandler(display)
```

Code Example 5-4 window.c (Continued)

```
     Display *display;
{
     exit(0);
}
```

Display PostScript (DPS) Overview

In the near future, SunSoft will be offering a new X server with Solaris 2.x which will utilize Display Postscript from Adobe Systems as its PostScript rendering engine. Display PostScript offers a device-independent imaging model which is fully compatible with the imaging model used in PostScript printers. SunSoft's Display PostScript implementation will be completely PostScript Red Book Level 2 compliant. Display PostScript components will include the Client Library (DPSlib), the PostScript interpreter, and the `pswrap` translator found in other Display PostScript implementations. In addition it will be offering Adobe's Display PostScript toolkit and resource database management libraries.

As with the other Solaris VISUAL libraries, Display PostScript relies on the calling X application program to provide the window area to draw into and to take care of managing the size of the window and window events. This allows the GUI portion of your application to be written in an X-based toolkit such as OLIT, XView, or Xlib, providing greater portability to platforms that also supply these toolkits and DPS. Inside the window is an area where you can display text, graphics, and images in a way that frees you from display-specific details such as screen resolution and the number of available colors usually dealt with through allocating colors either privately or from a global X color map. In this way there is a greater means for you to write code that can drive the display on the window and to print on a PostScript printer.

Comparing NeWS to Display PostScript

Display PostScript is analogous to what NeWS has been in past versions of OpenWindows. Both are similar in many ways; both offer ways to interpret PostScript and Display PostScript output in a device-independent fashion.

One can probably find the greatest similarity between NeWS and Display PostScript-based applications when NeWS is used through an extension to XView called `XVPS`. Like DPS, XVPS relies on an XView-based application to create a special canvas package called `PSCANVAS`, which is an extension to XView, and then XView would control window resizing and repaint events. Most applications using `XVPS` would also rely on XView to field input events as well as X events.

However, there are also many differences. NeWS affords greater control of the window system itself and input events which DPS does not attempt to use. DPS only provides a device-independent model for displaying information on a screen through OpenWindows. It does not try to be a window system itself.

NeWS has been PostScript Level 1-compatible in the past with some similar features to those found in PostScript Level 2. Now with Display PostScript, developers have the means to be completely compatible with the Level 2 standard which offers benefits such as increased functionality and performance. Some new features of PostScript Level 2 include: memory management and reclamation, resources, composite fonts, filters, binary encoding of objects and tokens, user paths, forms, color spaces, filters, more efficient graphics operators, improved halftoning, and stroke adjustment.

Display PostScript achieves greater performance through utilizing binary encoding offered in PostScript Level 2. Although Display PostScript can interpret ASCII PostScript code sent to it by using the binary encoding features of the Client Library and `pswrap`, the code that gets shipped down to the server is consistent and optimized so that the interpreter can speed up interpretation. NeWS offers binary encoding of its tokens as well. However, Display PostScript offers two forms of binary encoding: *binary tokens* and *binary object sequences*. Binary token encoding represents elements of the PostScript language as individual syntactic entities which emphasizes compactness over efficiency of generation or interpretation to aid in reducing bandwidth and storage space overhead. Binary object sequence encoding represents a sequence of one or more PostScript objects as a single syntactic entity. Instead of being designed for compactness as binary tokens are, it can be generated and interpreted quickly and therefore, most elements of the language are in a natural machine representation (or close to it) to allow one to gain performance where execution costs are relatively higher than other costs such as bandwidth and storage space.

Although using wrapped procedures can help with performance, not modularizing it enough won't buy much in performance. There are some types of functionality that are still better handled in compiled C code in the application client; for example, arithmetic computation or data manipulation which would be expensive when handled through an interpreter.

One other difference between DPS and NeWS (even when implemented using `XVPS`) is the way they both manage color. NeWS preallocates a segment of the X color map on 8-bit color machines as a static color map. This has the limitation of not being able to access the color map of another X application on the screen, it also needs to get the XID of other X clients it needs to get information on relative to other color information, for example the color cells of other X applications and the default color map. NeWS returns color values from the static color map that match closest to a requested HSB model color value for the `hsbcolor()` operator or RGB color model value for the `rgbcolor()` operator. Dithering of existing static color map values cannot be achieved unless the NeWS client process does it or you copy the contents of a 24-bit memory canvas to an 8-bit canvas. Not dithering sometimes produces banding effects which are more apparent the smaller the colorcube gets. DPS offers a new extension to support the cyan-magenta-yellow-black (CMYK) subtractive color specification for color printers and color separation devices, in addition to the RGB and HSB color operators that NeWS has. Also, DPS is able to dither colors rendered to use a closest-match algorithm, giving a closer approximation to the human eye of a range of colors than the closest match algorithm can achieve in 8-bit or less color environments.

SunSoft would like to encourage its NeWS and tNt developers to migrate their applications to the DPS model of using a standard OLIT, SunSoft's X toolkit which will receive the most support and enhancement for the GUI functionality of their application, and to utilize DPS for its ability to render PostScript imagery to facilitate the portability of their application to other X and DPS-based platforms.

DPS is able to support the execution of multiple PostScript language programs concurrently. Each PostScript execution context has its own independent thread of control, a set of stacks, private VM (virtual memory resource) or *space*, a shared VM, standard input and output files, and other miscellaneous state variables. Two or more execution contexts can actually be closely coupled and share the same *space* for applications such as text editors with multiple windows that need to have intimate knowledge of each others' states. Lock and condition semaphores are provided to manage this interaction and help synchronize their activity.

Note that when using DPS alongside other graphics libraries such as Xlib, especially those which utilize direct graphics access like XGL, you will need to synchronize Display PostScript output with the output of these graphics libraries. Since PostScript language programs run asynchronously on the X server, a PostScript rendering request is not guaranteed to be complete before a subsequent X request is executed, unless it is synchronized. You can handle synchronization either using wraps which return results and block until they are returned or by calling `DPSWaitContext` to trigger a wait. Waiting should be used only when the application has complete control of the PostScript code to be executed or with files that can be parsed and understood such as those following Adobe's Document Structuring Conventions. If an application cannot determine completeness of the code its executing, it should *freeze* a context using the `clientsync` function. In general, it is wise to try to minimize the amount of enforced synchronization as this can become an expensive of performance hit.

DPS Architecture

Components of Display PostScript include:

- the Client Library— call procedures from it to manage PostScript execution contexts
- the PostScript Interpreter—send programs and data to this and get results back
- the `pswrap` translator—creates wrapped procedures from PostScript language programs.

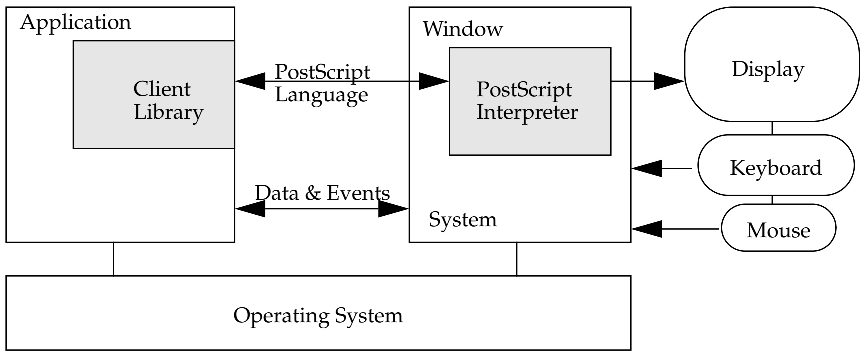

Figure 5-6 The Display PostScript System

Figure 5-6 shows how an application interacts with the Display PostScript system. The Client Library contains the data structures and procedures that the application will use to send request to the PostScript interpreter. The Client Library is analogous to what is known as the *cps* C client interface that has been used in NeWS-based programs in OpenWindows. You can write portable code with DPS to use the same procedure calls which are sent to the PostScript interpreter in the same way they are sent on other platforms by using the Client Library interface.

The DPS Client Library (libdps)

A large part of a DPS application is written in C or another high-level language. The Client Library is the application's primary interface to the Display PostScript System and is the only DPS application library available across all Display Postscript platforms. The DPS Client Library on OpenWindows Version 4 is implemented as an X extension to the window system. For NeWS developers, it can be thought of as being similar in functionality to the `libcps` library used in NeWS. Like `libcps` it consists of a set of C routines which generate PostScript code that is sent to the PostScript interpreter resident in the X server for execution.

The Client Library contains procedures for creating, communicating with, and destroying PostScript execution contexts. A context consists of all of the "state" information needed by the PostScript interpreter to execute a PostScript Language Program. To the application programmer it appears that the C Client Library calls trigger the rendering themselves and would, therefore, allow one to develop an application entirely in C. The Client Library maps the PostScript operators map completely when using single-operator procedures defined in `dpsops.h` and `psops.h`. However, application developers will find that they can achieve better performance and greater efficiency with less overhead by creating *wrapped* procedures in a modular fashion to use in addition to the single-operator procedures offered by the Client Library.

In general the phases of an application using the Client Library consist of:

- **Initialization**
  This phase sets up communication with the Display PostScript system in the X server. This is done by connecting through Xlib's `XOpenDisplay()` or a toolkit such as OLIT's standard initialization process. This step will define an *X Display* record for the connection. After that an *X drawable* (window or pixmap) and an X GC must be created through Xlib or the window toolkit.

- **Creating a Context**
  The program then uses Client Library calls to create a context for executing PostScript programs through either `XDPSCreateSimpleContext()`, which uses the default color map, or `XDPSCreateContext()`,which allows one to specify color map information. The application provides a color map with a uniform distribution of colors in the form of a *grayramp* for grayscale devices or a *ccube* (color cube) for full color devices. If you want to conserve color map entries or display pure (non-dithered) colors in the way other X applications commonly do, the argument `actual` can be added as a hint to the server to reserve a number of color cells which get allocated on a first-come first-served basis to the application when the application executes a `setrgbcolor` instruction.

- **Execution**
  Once initialized and contexts have been established, PostScript graphics are rendered by the PostScript language programs being sent to the interpreter in the X server. The Client Library sends these programs and handles the results received back from the interpreter.

- **Termination**
  Once the application is ready to terminate, it calls the Client Library to destroy its contexts, free their resources, and end the session.

pswrap

The `pswrap` translator provides a convenient way for developers to create C-callable procedures that contain sets of PostScript code which are composed by the application developer. The procedures are known as *wrapped procedures* or *wraps*. The *wrap body* contains the PostScript language program segment contained in the *wrap*.

The process for defining and using a *wrap* through `pswrap` in Display PostScript is:

- Write the PostScript programs using `pswrap` syntax with a C-callable procedure and input and output arguments where needed.
- Run `pswrap` to translate these PostScript language programs into wrapped procedures.

- Compile and link these wraps with the application program.
- When a wrap is called, the PostScript code associated with it is sent to the interpreter and returns values returned by the interpreter.

Input arguments to wraps are values sent to the PostScript interpreter as PostScript objects; output arguments are pointers to variables where the wrap stores values returned by the PostScript interpreter.

The syntax of a wrapped procedure consists of four parts:

- `defineps`
  Begins the definition of a wrap and must be at the beginning of the line without any preceding space or tab characters.
- Declaration of the C-callable procedure
  The name of the C procedure with its parameters in parenthesis. The parameters themselves are optional but the parentheses are not. There is no value returned by a wrap, and, therefore, it should be declared `void`. Note that if output parameters are in an argument list, they follow the input parameter list and are preceded by a single vertical bar character.
- the wrap body
  PostScript Language program fragment which is sent to the interpreter.
- `endps`
  Ends the definition; like `defineps`, `endps` must appear at the beginning of the line.

The following code example shows what a wrap looks like in a file denoted conventionally with a `.psw` extension.

```
defineps PSWDisplayText(float X,Y; char *text)
    X Y moveto
    (text) show
endps
```

The call to this wrap in a C program looks as follows:

```
PSWDisplayText(72.0, 100.0, "Hello World");
```

The following code example shows a wrap with a output parameters defined:

```
defineps PSW_bbox(| int *x0,int *y0,int *x1,int *y1)
    clippath pathbbox y1 x1 y0 x0
endps
```

The corresponding call in the C program to retrieve the bounding box would be:

```
long int x0, y0, x1, y1;
PSW_bbox(&x0, &y0, &x1, &y1);
```

If you have programmed with NeWS, you might see a similarity between *pswrap* and the *cps* utility which compiles `cps` macros in a similar way to produce C-callable entry points to your C program. The following code segments duplicate the declarations of the functions in the preceding DPS *wrap*s but use the NeWS `cps` structure that would have been used in a `.cps` file before it was processed by `cps`.

In the first example note the only difference is that `cdef` is used instead of `defineps` and that there is no terminating statement such as `endps`.

```
cdef PSWDisplayText(float X, float Y, char *text)
    X Y moveto
    (text) show
```

Note that the C call from the NeWS client would look the same as that in the DPS example:

```
PSWDisplayText(72.0, 100.0, "Hello World");
```

Note that the way `cps` in NeWS works with output parameters is slightly different. Instead of separating input and output parameters in the C function argument list by a vertical bar, the procedure and argument list in parentheses is followed by a => sequence and then a *tag* identifier followed by a parenthesized list of output result fields where the resulting values are returned. Note that every result argument must be identified and declared in the preceding function argument list.

To achieve synchronization, the constant tag value in the example below, `57`, which the client is waiting for to retrieve the coordinate results from the server, is sent first with a `tagprint` operator. Following this the coordinate values are popped off the stack in reverse order (which is why the results field is in reverse order) and sent back to the client with the `typedprint` operator.

A NeWS `cps` macro using output parameters:

```
#define BBOX_TAG 57
cdef PSW_bbox(int x0,int y0,int x1,int y1) => BBOX_TAG(y1, x1, y0, x0)
    clippath pathbbox

    BBOX_TAG tagprint
    typedprint
```

```
    typedprint
    typedprint
    typedprint
```

As in the DPS example, the corresponding call in the C program to retrieve the bounding box would be:

```
long int x0, y0, x1, y1;
PSW_bbox(&x0, &y0, &x1, &y1);
```

In DPS, `printobject`, an operator similar to the NeWS `typedprint` operator is used to return values in the form of binary object sequences from the interpreter, and precedes each value returned with a tag and terminates the binary sequence with a special termination tag. In DPS, the `printobject` calls are not put in the wrapped procedures written by the application developer but are generated as part of the output when they are run through `pswrap`.

Also with the Client Library for DPS you can put arrays in as arguments whereas you can not with NeWS's `cps`.

Note that there are a number of other differences between NeWS and DPS, but you will observe that if you have existing NeWSbased applications that tend to be more output-oriented, talk to an X drawable as is done through `xvps`, focus more on rendering than a lot of window-based functionality, and use the `cps` model rather than a NeWS toolkit, such as tNt, porting such applications to DPS should be fairly straightforward because, for the most part, the PostScript code will stay the same, and `cps` declarations just need to be altered to utilize the syntax accepted by `pswrap`.

Font Support

The good news about SunSoft's font support with Display PostScript is that you get support for not only the F3 Folio scalable fonts supported in OpenWindows in the past, you also get full support of Adobe Type 1 scalable fonts. Therefore, you will be able to use the many Adobe Type 1 font sets available, as well as the set of F3 scalable fonts that have been present in OpenWindows.

DPS Toolkit (libdpstk)

In addition to the Client Library for DPS, there is also the Display PostScript toolkit that consists of a collection of procedures and objects, which could be considered "convenience routines" to the Client Library. This toolkit can be used for context management, user object management, user path handling, and file previewing. SunSoft's version of this toolkit is being implemented to integrate well with developers using OLIT as their X toolkit. Keep in mind that the DPS toolkit is X specific and is not necessarily offered by all Display PostScript vendors, so developers interested in portability to other DPS platforms might want to keep their development efforts focused on using the Client Library instead.

In the case of context management, the DPS toolkit helps manage PostScript execution contexts by making it easier for different X drawables to share one Display PostScript context, different code modules to share contexts, and hide the details of context creation from the application by creating and managing default contexts.

In addition to providing procedures to conveniently manage user objects and user paths, the DPS toolkit provides procedures for simplifying rendering of PostScript language files into X drawable objects. There are also a number of procedures that facilitate handling of font selection and preview from within applications. See the *Display PostScript Toolkit for X*, which is part of the *Display PostScript System: Reference Manuals* from Adobe Systems, for more information on the DPS toolkit.

DPS Resources and the Resource Library (libpsres)

Applications need to locate to a set of files that contain descriptions of PostScript objects. These objects can be Adobe font metric files (AFM files), font outline files, PostScript Language procedure sets, forms, patterns, encoding, or any named PostScript object; these are collectively known as *resource files*. Given that these files can be available to a single user or system-wide for uses such as fonts that reside on a system's PostScript printer, to effectively manage access to the resources in the many locations they could be, *resource database files* are necessary to help applications locate resources uniformly.

The format of a resource database file consists of the following components:

- **Identifying String** (required)
  This must be either of the constant strings `PS-Resource-1.0`, if there are more resource database files or files with filename suffix `upr` in the same directory besides the default principle resource database file, `PSres.upr`, or `PSResources-Exclusive-1.0` if the principal resource database file, `PSres.upr`, is the only resource database file in its directory. The latter will help resource location procedures run more efficiently.
- **Resources Type Component** (required)
  This lists the resource types described by the file, one per line, with the final line containing a period indicating the end of the component.
- **Directory Component** (optional)
  This component is a single line that identifies the directory prefix to be added to all filenames in the resource database. Not specifying this parameter assigns the directory of the resource database file as the directory prefix.

- **Resource Data Components** (required)
  Each resource type requires a data component specified in the same order as their identifiers in the resources type component. Each data component consists of one line for the resource type identifier, followed by lines of resource data, followed by a line with a terminating period. Each line of resource data contains the name of the resource, a single or double equal sign, and the name of the file which contains the resource.

There are a set of predefined resources that describe fonts (`FontOutline`, `FontPrebuilt`, `FontAFM`, `FontBDF`, `FontBDFSizes`, and `FontFamily`) and other resources such as `Form`, `Pattern`, `Encoding`, `ProcSet`, and `mkpsresPrivate`. To specify where procedures look for PostScript resource database files, one should set the `PSRESOURCEPATH` variable to point to the directories where they reside.

The library, `libpsres`, contains procedures that locate and parse resource database files and returns lists of resource files. `makepsres` is a utility that can be used to create a resource database file containing resources in all directories specified on the command line and any resources it may find in resource database files that already exist in those directories.

See Appendix A and B of *The Display PostScript Toolkit for X,* which is part of the *Display PostScript System: Reference Manuals* from Adobe Systems, for more information on resource usage and using the `makepsres` utility.

A DPS Program

The following program demonstrates a simple example on how to use the Display PostScript system through the Client Library. The example:

- establishes communication with an X11 server
- creates a window and a context
- draws an ochre rectangle in the window
- waits for a mouse-button click
- terminates when the button is pressed

Use the pswrap program to generate a C code file and C header file from the wrap declared in the example examplewraps.psw which follows example.c. (These examples are contained in the *Display PostScript System: Reference Manuals,* Adobe Systems Incorporated.)

Code Example 5-5 example.c

```
/*
 example.c - simple X Window System application. Uses Display PostScript
 to draw an ochre box and uses X primitives to wait for a mouse click before
 terminating.
*/

#include <stdio.h>            /* Standard C library I/O routines */
#include <string.h>           /* Standard C library string routines */
#include <X11/X.h>            /* X definitions */
#include <X11/Xlib.h>         /* Interface to X library */
#include <X11/Intrinsic.h>    /* X toolkit definitions */
#include “psops.h”            /* Interface to PostScript single-op wraps */
#include “dpsXclient.h”       /* Interface to the DPS Client Library */
#include “examplewraps.h”     /* Interface to user-defined
                              “wrap” procedures */

/* Window geometry definitions */
#define XWINDOW_X_ORIGIN  100
#define XWINDOW_Y_ORIGIN  100
#define XWINDOW_WIDTH     500
#define XWINDOW_HEIGHT    500

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

{
 Display *dpy;         /* X display structure */
 int screen;           /* screen on display */
 DPSContext ctxt;      /* DPS drawing context */
 DPSContext txtCtxt;   /* DPS text context for debugging */
 Window xWindow;       /* window where drawing occurs */
 int blackPixel, whitePixel;
 int debug = { FALSE };
 GC gc;
 XSetWindowAttributes attributes;
 unsigned long mask;
 DPSSpace space;
 float x, y, width, height;
```

Code Example 5-5 example.c (Continued)

```
/* Connect to the window server by opening the display. Most of command
line is parsed by XtOpenDisplay, leaving any options not recognized by
the X toolkit; look for local -debug switch */

XtToolkitInitialize();
dpy = XtOpenDisplay(NULL, (String) NULL, "example", "example",
      (XrmOptionDescRec *) NULL, 0, &argc, argv);
screen = DefaultScreen(dpy);
if (argc == 2)
   if (strcmp(argv[1], "-debug") == 0)
       debug = TRUE;
   else {
       printf("Usage: example [-display xx:0] [-sync] [-debug]\n");
exit(1);
}
/* Create a window to draw in; register interest in mouse button events. */

blackPixel = BlackPixel(dpy, screen);
whitePixel = WhitePixel(dpy, screen);
attributes.background_pixel = whitePixel;
attributes.border_pixel = blackPixel;
attributes.bit_gravity = SouthWestGravity;
attributes.event_mask = ButtonPressMask | ButtonReleaseMask;
mask = CWBackPixel | CWBorderPixel | CWBitGravity | CWEventMask;
xWindow = XCreateWindow(dpy, DefaultRootWindow(dpy),
XWINDOW_X_ORIGIN, XWINDOW_Y_ORIGIN, XWINDOW_WIDTH, XWINDOW_HEIGHT,
   1, CopyFromParent, InputOutput, CopyFromParent, mask, &attributes);

XMapWindow(dpy, xWindow);

gc = XCreateGC(dpy, RootWindow(dpy, screen), 0, NULL);
XSetForeground(dpy, gc, blackPixel);
XSetBackground(dpy, gc, whitePixel);

/* Create a DPS context to draw in the window we just created. If the
user has asked for debugging, create a text context chained to the
'drawing' context. */

ctxt = XDPSCreateSimpleContext(dpy, xWindow, gc, 0, XWINDOW_HEIGHT,
DPSDefaultTextBackstop, DPSDefaultErrorProc, NULL);
if (ctxt == NULL) {
   fprintf(stderr, "Error attempting to create DPS context\n");
   exit(1);
}
```

Code Example 5-5 example.c (Continued)

```
 DPSSetContext(ctxt);

 if (debug) {
   txtCtxt =
       DPSCreateTextContext(DPSDefaultTextBackstop,DPSDefaultErrorProc);
   DPSChainContext(ctxt, txtCtxt);
 }

 /* Convert the X Window System coordinates at the lower right corner
 of the window to get the width and height in user space. */

 PSitransform(
   (float) XWINDOW_WIDTH,
   (float) -XWINDOW_HEIGHT,
   &width,
   &height);

 /* Locate the box in the middle of the window. */

 x = width / 4.0;
 y = height / 4.0;

 /* Paint an ochre box. */

 PSWDrawBox(0.77, 058, 0.02, x, y, width / 2.0, height / 2.0);

 /* Wait for a mouse click on any button then terminate */

 while (NextEvent() != ButtonPress);
 while (NextEvent() != ButtonRelease);

 space = DPSSpaceFromContext(ctxt);
 DPSDestroySpace(space);
 exit(0);

} /* main */

int NextEvent()
{
 XEvent    event;

 XtNextEvent(&event);
 return(event.type);
}
```

Wraps for the `example.c` program are declared in the following file; compile this with `pswrap`.

Code Example 5-6 examplewraps.psw

```
/* wrap for example application */

defineps PSWDrawBox(float r, g, b, x, y, width, height)
 gsave
 r g b setrgbcolor
 x y width height rectfill
 grestore
endps
```

DPS Summary

Doing a complete comparison between NeWS and tNt toolkits and DPS is beyond the scope of this book. However, a general rule of thumb to use while porting is to move GUIoriented code from tNt or NeWS to an X-based toolkit, and then use DPS to do the rendering of imagery and items such as font selection boxes.

For more information on Display PostScript, please see the *Display PostScript System: Reference Manuals* in the bibliography at the end of this chapter and look at the technical notes from the *Adobe Developers' Association*. They can be reached at the following address:

PostScript Developer Support
Adobe Systems Incorporated
1585 Charleston Road, P.O. Box 7900
Mountain View, CA 94039-7900

Email file server: **ps-file-server@adobe.com** with “Help” in the subject line.

Phone: **(415) 965-4111**.

Xlib and Direct Xlib

Solaris has shipped, and will continue shipping, the industry-standard Xlib library. Many ISVs choose Xlib as a graphics rendering toolkit for graphic rendering that tends to be 2D and low or medium performance. ISVs wishing to find ways of increasing graphics performance but maintaining portability with other X-based platforms may choose to use Direct Xlib. For the first time Direct Xlib 2.0 is now bundled with Solaris 2.x and works with OpenWindows version 3.1.

New features of Direct Xlib 2.0 include:

- simplified installation
- native SunOS 5.2 support
- X11R5 support
- I18N level 4 support
- use of shared fonts for all Xlib text functions
- accelerated 16-bit fonts
- loadable graphics driver support

Dynamically linked applications can switch libraries to point to Direct Xlib libraries without recompilation and realize performance gains. This is the recommended usage of Direct Xlib.

Static applications can also be relinked against the Direct Xlib libraries, but they will not run on servers where Direct Xlib is not active. Applications that are statically linked must link to both Direct Xlib's static `libX11` library and to the math library (`-lm`). An application that statically links to `libXext` or `libXinput` must statically link to Direct Xlib's `libXext` or `libXinput` and to Direct Xlib's `libX11`.

Applications that stand to benefit most from Direct Xlib tend to be those with large numbers of short vectors, filled rectangles, short text strings, and small scrolls or bit operations where the overhead of copying X11 commands and data into the Xlib request buffer and interprocess communication is a significant amount of the processing. Also note that unless used with a supported graphics accelerator, Direct Xlib will work with graphics devices such as bw2 and cg3 as with standard X, but will not provide any performance improvement; they may present some extra overhead from the Direct Xlib layer in between.

Run `/opt/SUNWdxlib/bin/dxlib_status` to verify that Direct Xlib is active. The `-v` option provides additional information.

PEX and PEXlib

PEX (which originally stood for "PHIGS Extensions to X") is a protocol extension to the MIT X11 Network Window System that was originally designed to provide efficient support for PHIGS and PHIGS PLUS. PEX allows graphical applications to display across the network. When PHIGS programs are run remotely across the network, PEX allows the developer to tune performance by having PHIGS display list structures and associated information be stored either on the server or the remote client, depending on which provides the best performance when objects are rendered on the server's display.

The runtime environment of SunPEX™ 2.0, SunSoft's latest implementation of PEX, is now bundled with Solaris 2.x. The developer's environment is available unbundled and as a part of the Sun Developer's Kit. SunPEX 2.0 differs from SunPEX 1.0 in one major respect. In Figure 5-7, XGL is underneath the `ddpex` (device-dependent PEX layers) of the OpenWindows PEX extension. Therefore whatever devices are supported by XGL will in turn also be supported by SunPEX 2.0, offering third-party display device manufacturers and Solaris OEMs a better way to get access to PEX on Solaris platforms and ISVs more display options for their applications.

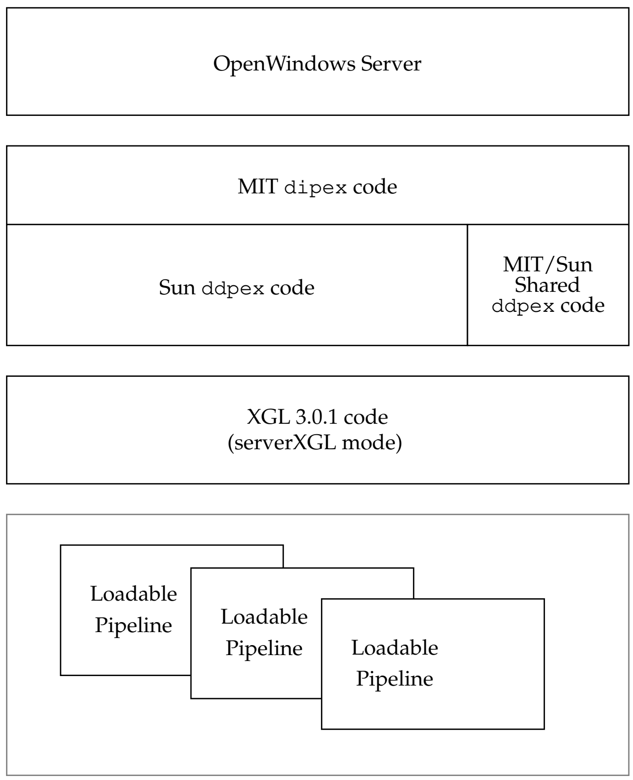

Figure 5-7 SunPEX 2.0 Server Architecture

SunPEX 2.0 also provides support for and inclusion of PEXlib, the emerging low-level standard 3D API supported by Sun, IBM, DEC, HP, Kubota, Tektronix, NCD, and Showgraphics. Based on the PEX 5.1 specification from MIT, the Client Library, PEXlib, is closer to the PEX protocol than PHIGS and also supports immediate mode, which the PHIGS library does not support and which some applications need. If you need to develop PEXlib-based applications, see the *PEXlib Programming and Reference Manual*s.

Added features that SunPEX 2.0 provides for not found in the PEX-SI from MIT include:

- Z-buffering, double buffering
- more fonts

- hollow polygons, multi-bounded polygons
- dithering, color interpolation (vs. flat-shading)

Extensions included in SunPEX 2.0 include:

- Circle, CircArc, CircArcCLose
- AnnotCircle, AnnotCircArc, AnnotCircArcClose
- Ellipse, EllipArc, EllipArcClosed
- AnnotEllipse, AnnotEllpArc, AnnotEllpArcClosed
- RectGrid, RadialGrid
- TriangleList
- HighlightColour
- TextSlantAngle, AnnotTextSlantAngle
- StrokeAAliasParams
- TextStrokeWidth
- EndCap, StrokeJoin
- SilhouetteEdge
- SurfTransp
- BufferMode
- DefineMarkerType
- ExtRendAttr
- Line Type
- Hatch Styles
- "E & S" (Evans and Sutherland) Sphere and Cylinder family
- other "E & S" extensions

To help you determine whether PEX is running, and to help with any other problems that might occur, there are two utilities, `pexdpyinfo` and `pexscope`, that are similar to their X counterparts, `xdpyinfo` and `xscope`.

`Xdpyinfo` detects whether the PEX extension `X3D-PEX` is installed in the active server along with all of the other extensions. `pexdpyinfo` provides information about the installed PEX extension, such as the supported enumerated types. `pexscope` lets you debug PEX protocols going over the wire by intercepting PEX events much like `xscope` does with X events.

Note that PEX is not PHIGS or PEXlib-specific; other APIs can be made to talk to the PEX protocol and can benefit from the distributed PEX protocol. XGL, and, therefore, the APIs layered upon it, also supports PEX and uses the same distributed PEX protocol that PHIGS/PEX and PEXlib do. A good description of PEX can be found in *Advances in Computer Graphics 6*.

Note – To run PEX, you should have a minimum of 16 MB of memory installed; 32 MB is preferable. You should also add 50 MB of swap space over and above the amount of swap space normally allocated for Solaris 2.x.

Some environment variables that are useful are `$PEXLIBHOME`, which tells PEX where to find its runtime libraries, etc., if they are different from `$OPENWINHOME/lib`, and `$PHIGS_LOCAL_USE_PEX`, which lets you override the PHIGS client's direct usage of DGA and have the client use PEX instead for locally running applications.

Bibliography

Advances in Computer Graphics 6, G. Garcia and I. Herman, Springer-Verlag.

Display PostScript Programming, David Holzgang, ISBN 0-201-51814-7, Addison-Wesley Publishing Company.

Display PostScript System: Reference Manuals, Adobe Systems Incorporated.

NeWS 3.0 Programmer's Guide, Part Number 800-6736-10, Rev. A, SunSoft, Inc., Mountain View, CA.

PEXlib Programming Manual, O'Reilly & Associates.

PEXlib Reference Manual, O'Reilly & Associates.

PHIGS Programming Manual, Tom Gaskins, ISBN 0-937175-85-4, O'Reilly & Associates.

PHIGS Reference Manual, O'Reilly & Associates.

A Practical Introduction to PHIGS/PHIGS Plus, T. Howard et al, Addison-Wesley Publishing Company.

A Primer for PHIGS, F. Hopgood and D. Duce, John Wiley & Sons, inc..

The PostScript Language Tutorial and Cookbook, Adobe Systems Incorporated, ISBN 0-201-10179-3, Addison-Wesley Publishing Company.

The PostScript Language Reference Manual, 2nd Edition, Adobe Systems Incorporated, ISBN 0-201-18127-4, Addison-Wesley Publishing Company.

The PostScript Language Program Design, Adobe Systems Incorporated, ISBN 0-201-14396-8, Addison-Wesley Publishing Company.

Running IRIS GL Applications on SPARCstations Using Nth Portable GL, Part Number 801-2022-05, Sun Microsystems, Inc., Mountain View, CA

Solaris XGL 3.0 Programmer's Guide, Part Number 800-7691-10, SunSoft, Inc., Mountain View, CA.

Solaris XGL 3.0 Reference Manual, Part Number 800-7692-10, SunSoft, Inc., Mountain View, CA.

Solaris XIL 1.0 Imaging Release Notes, Part Number 801-2022-10, SunSoft, Inc., Mountain View, CA.

Solaris XIL 1.0 Imaging Library Programmer's Guide, Part Number 801-2020-10, SunSoft, Inc., Mountain View, CA.

Solaris XIL 1.0 Imaging Library Reference Manual, Part Number 801-2021-10, SunSoft, Inc., Mountain View, CA.

XGL-3.0 Technology Brief, Part Number 922-61-001, SunSoft, Inc., Mountain View, CA.

XGL 3.0 Device Pipeline Porting Guide, Part Number 800-7693-01, SunSoft, Inc., Mountain View, CA.

XGL 3.0 Architecture Specification-Draft Version 1.4, Part Number 800-7690-05, SunSoft, Inc., Mountain View, CA.

Solaris XGL Graphics Library: A Technical White Paper, Part Number 93047-001, SunSoft Inc., Mountain View, CA.

XView Programmer's Manual, ISBN 0-9317175-87-0, O'Reilly & Associates.

X Window System Programmer's Supplement to the Client Library Reference Manual, Adobe Systems, Inc.