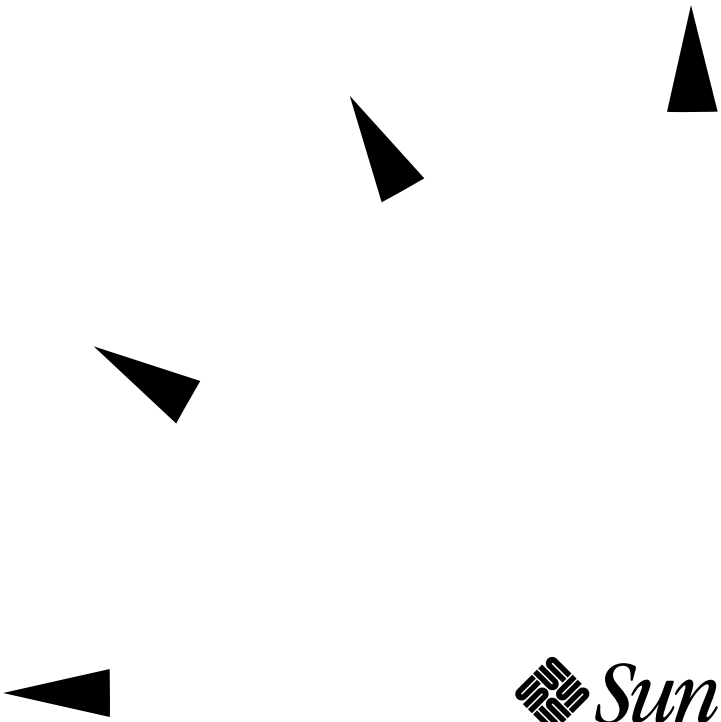
Getting Started Writing XGL Device Handlers



SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Please
Recycle

Adobe PostScript

Contents

Figures

Tables

Preface

The *Getting Started Writing XGL Device Handlers* manual explains how to use the XGL™ skeleton pipeline files to create an XGL graphics handler for a graphics hardware device.

Who Should Use This Manual

This manual is intended for implementors of XGL graphics handlers. It is assumed that you are is familiar with the C and C++ languages.

How This Manual Is Organized

This manual is organized as follows:

Chapter 1, “Introduction to the Skeleton Pipeline,” describes the skeleton pipeline and provides a brief overview of the XGL architecture.

Chapter 2, “Building the Reference Pipelines,” documents how to build the sample XGL graphics handlers provided with the XGL Driver Developer Kit.

Chapter 3, “Implementing the Skeleton Pixel-Level Graphics Handler,” explains how to modify the skeleton source files to create a pixel-level XGL graphics handler for your device.

Chapter 4, “Implementing Accelerated Primitives,” provides basic information on how to modify the skeleton source files to implement accelerated primitives for your device.

Related Manuals

For information on the XGL architecture and the design of the loadable pipelines, see the following manual:

- *XGL Architecture Guide*

For information on the XGL test suite, see:

- *XGL Test Suite User's Guide*

For information on the XGL library, see:

- *XGL Reference Manual*
- *XGL Programmer's Guide*
- *XGL Accelerator Guide for Reference Frame Buffers*

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this manual.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.

Introduction to the Skeleton Pipeline 1

About the Skeleton Pipeline

The Solaris Device Developer's Kit (DDK) includes a template for an XGL™ graphics handler. This template, called the skeleton pipeline, is provided partially implemented and will help you get started with your implementation of an XGL graphics handler.

An XGL graphics handler consists of a set of loadable renderers that send geometry data to the hardware, and a set of interface objects that provide communication between the XGL device-independent code and the device pipeline code. The pipeline interface objects form a framework that connects the device-independent code with the device pipeline rendering code. Figure 1-1 illustrates the basic components of the graphics handler.

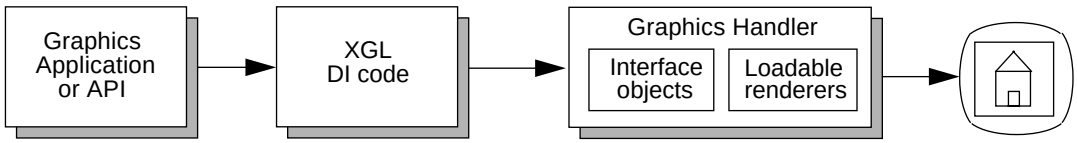


Figure 1-1 High Level View of the XGL Architecture

The skeleton pipeline includes partially implemented files for the device pipeline interface objects; these files can expedite the implementation of your device pipeline. The skeleton pipeline also provides a simple implementation of line and polygon accelerated renderers that you can implement for your device.

1

Overview of XGL Architecture

The XGL library contains two primary components: an application programming interface (API) for application developers and a graphics porting interface (GPI) for hardware vendors. The XGL GPI is a device-level interface that defines the mapping of XGL device handlers to underlying hardware. Hardware vendors that write XGL device handlers can build graphics devices that support any binary XGL application.

The XGL GPI consists of three layers of device pipeline interfaces. Each layer defines a set of rendering tasks that must be accomplished before proceeding to the next layer in the pipeline. More complex operations, such as transformations, lighting, and clipping, are performed in the uppermost layer; less complex operations, such as scan conversion, are performed in the lower layers.

The top layer of the GPI, Loadable Interface 1 (LI-1), specifies the interface that lies directly below the XGL API. Functions in this layer take the points defining the primitive and transform, light (in the 3D case), and clip the geometry in preparation for the rendering operations in the next layer. The second layer (LI-2) is responsible for scan converting more complex primitives like polygons and polylines. The third layer (LI-3) is responsible for rendering pixels, individually or in spans on the device. The GPI includes a complete software implementation of the LI-1 and LI-2 layers of the pipeline for most primitives; however, the lowest layer, which is responsible for writing pixels to the device, is device dependent and is not included in the software implementation.

Device pipelines written at the LI-1 layer typically implement the full graphics pipeline for each primitive, including all LI-1 operations, scan conversion, and pixel rendering. Device pipelines written at the LI-2 layer call the software pipeline for LI-1 operations and then take over processing at LI-2, performing scan conversion and rendering pixels on the device. LI-3 device pipelines are responsible only for rendering pixels; a port at this layer uses the software pipeline for LI-1 and LI-2 operations. Hardware vendors can implement different GPI functions at different layers to tailor a port for a particular device.

Figure 1-2 illustrates the layers of the device pipeline and software pipeline and some components of the XGL device-independent code.

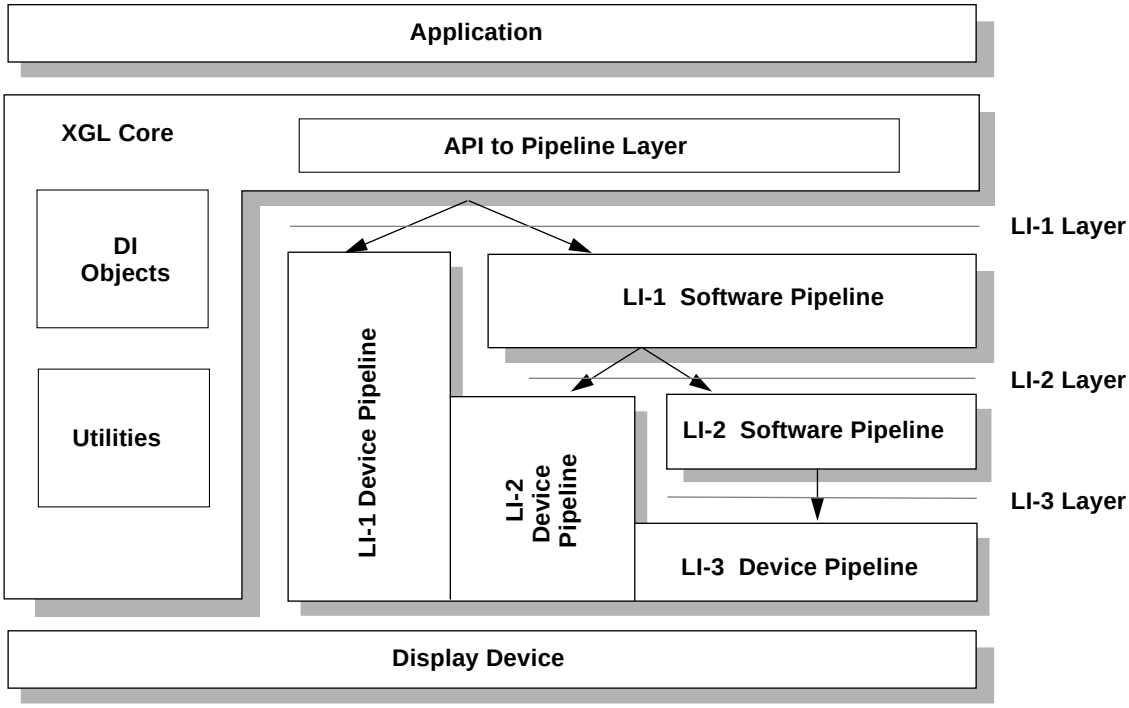


Figure 1-2 XGL Graphics Porting Architecture

For more information on the XGL architecture and for comprehensive information on implementing an XGL graphics handler, see the following manuals:

- *XGL Architecture Guide*
- *XGL Device Pipeline Porting Guide*
- *XGL Test Suite User's Guide.*

For information on the XGL library, see the following:

- *XGL Reference Manual*
- *XGL Programmer's Guide*

1

Design Considerations

Before you begin developing your graphics handler, you need to determine how you will handle several important design issues:

- What interface layer (LI-1, LI-2, or LI-3) will you port to? A graphics handler can include routines at all three layers, but usually the characteristics of the device will determine what interface layer is the primary porting layer.
- Which attributes and primitives can your hardware accelerate?
- Which attributes and primitives are needed by the kind of applications you are targeting? What features would those applications like to have accelerated?
- How will your graphics handler support multiple XGL contexts?
- How will your graphics handler support backing store?
- Do you need to port Direct Graphics Access (DGA) for your device?

For information on these issues, refer to the *XGL Device Pipeline Porting Guide*.

Files Provided With the Skeleton Pipeline

Table 1-1 lists the files for the skeleton pipeline and provides a brief description of these files. For instructions on editing these files to build a pixel-level graphics handler, see Chapter 3. For instructions on editing the sample renderers, see Chapter 4.

Table 1-1 Skeleton Source Code Files

File	Description
`DpLibSkeleton.h` `DpLibSkeleton.cc`	Header and source files for the XglDpLib object. This object is created at driver installation time. It creates one or more XglDpMgr objects.
`DpMgrSkeleton.h` `DpMgrSkeleton.cc`	Header and source files for the XglDpMgr object. This object manages hardware initialization and creates the XglDpDev object. Typically, there is one XglDpMgr object per hw device.

Table 1-1 Skeleton Source Code Files *(Continued)*

File	Description
`DpDevSkeleton.h` `DpDevSkeleton.cc`	Header and source files for the XglDpDev object. This object corresponds to the XGL window raster and creates the XglDpCtx objects.

`DpCtx2dSkeleton.h`
`DpCtx2dSkeleton.cc`
`DpCtx3dSkeleton.h`
`DpCtx3dSkeleton.cc`
	Header and source files for the 2D and 3D XglDpCtx objects. These objects are created by the XglDpDev object once for every XGL context-raster association. The XglDpCtx objects contain the interfaces for the 2D and 3D primitives.

`PixRectSkeleton.h` `PixRectSkeleton.cc`	Header and source files for a device-specific PixRect object. Memory mapped devices do not need this object. If the device is not memory mapped, or only one buffer can be accessed at a time, this object is needed for pixel rendering.
`Skeleton2dLi3.cc` `Skeleton3dLi3.cc`	Source files that contain the rendering routines for LI-3 primitives.
`Skeleton2dLi1Raster.cc` `Skeleton3dLi1Raster.cc`	Source files that contain the LI-1 raster functions..
`Skeleton2dLi2Pgon.cc`	Source file for an LI-2 accelerated polygon renderer.
`Skeleton3dLi1Mpl.cc`	Source file for an LI-1 accelerated multipolyline renderer.

1

Building the Reference Pipelines 2

This chapter describes how to build the reference XGL device pipelines provided with the XGL DDK product. Although building the reference pipelines is not required, you may want to build one or more pipelines to check that the DDK product and the compilers have been properly installed.

Reference Pipelines Provided with the XGL DDK

Source code for the reference pipelines listed in Table 2-1 is provided with the XGL DDK as examples of XGL device pipelines. The reference pipelines are located in `/opt/SUNWddk/ddk_2.5.1/xgl/src/dd`.

Table 2-1 XGL Reference Pipelines

Name	Description	Features
`cfb`	SPARC simple color frame buffer pipeline (CG3/CG8)	8-bit or 24-bit unaccelerated DGA rendering. Uses RefDpCtx for rendering.
`cg6`	SPARC TGX frame buffer pipeline	Example of an LI-1 8-bit color DGA pipeline. Accelerates 2D solid and patterned lines, simple polygons without holes, and multisimple polygons. Accelerates 3D solid and patterned lines, solid simple and multisimple polygons, and flat-shaded, non-Z buffered triangle strips.
`cgm`	CGM pipeline	Outputs CGM formatted files.

Table 2-1 XGL Reference Pipelines *(Continued)*

Name	Description	Features
`mem`	Memory pipeline	Renders primitives to host memory.
`p9000`	x86 pipeline	Example of an LI-2 8-bit pipeline. Accelerates 2D and 3D lines and 3D multisimple polygons, regular polygons, triangle strips, and quad meshes. Also accelerates LI-3 spans.
`p9100`	x86 and PowerPC™ pipeline	Example of an LI-2 8-bit pipeline. Accelerates 2D and 3D lines and 3D multisimple polygons, regular polygons, triangle strips, and quad meshes. Also accelerates LI-3 spans.

Software Prerequisites

Before you build the XGL reference pipelines, you must have the Solaris environment installed. In addition, the XGL runtime and XGL SDK packages must be installed. If these packages have been installed to the default locations, you can determine whether the necessary files are available as follows:

- For the XGL runtime `libxgl.so`, check the directory `/opt/SUNWits/Graphics-sw/xgl/lib`.
- For the XGL SDK include files `xgl*h`, check the directory `/opt/SUNWsdk/sdk_2.5.1/xgl`.

Note – The `cg6` pipeline depends on header files that are included in the Solaris Sample Device Drivers package (`SUNWDrvs`). This package must be installed on your system before the `cg6` pipeline can be built.

Note – In Solaris 2.5.1, the XGL include files do not reside in the same directory as the runtime binary. However, a symbolic link from `/opt/SUNWits/Graphics-sw/xgl/include/xgl` to `/opt/SUNWsdk/sdk_2.5.1/xgl/include/xgl` is created when the SDK packages are installed. As a result, the `XGLHOME` environment variable works as in previous releases.

Space Requirements

The reference pipelines can require up to 4 Mbytes of disk space, depending on the number of pipelines you build. Table 2-2 shows the approximate amount of space needed for each pipeline.

Table 2-2 Pipeline Space Requirements

Pipeline	Size
mem	305K
cfb	272K
cg6	2.3 Mbtyes
cgm	580K
p9000	500K
p9100	500K

Steps for Building the Reference Pipelines

Follow the steps below to build the reference pipelines.

1. **As super-user, run the** `change_owner` **script to change owner on the** `include`**,** `src`**, and** `lib` **directories so that the directories are owned and writable by the user who will perform the build.**

```
# <DDK_DIR>/bin/change_owner [owner] [group]
```

2. **Exit root login and become user.**

3. **Set the** `$XGLHOME` **environment variable to point to the directory where the XGL runtime and include files are located.**
   The default installed location is `/opt/SUNWits/Graphics-sw/xgl`.

```
% setenv XGLHOME <XGL runtime directory>
```

4. **Change directory to the directory where the XGL DDK package is installed.**
   The default location is `/opt/SUNWddk/ddk_2.5.1/xgl`.

```
% cd <DDK_DIR>
```

5. **Run the `setup_links` script.**
   The `setup_links` script adds links in the DDK `lib` directory that point to the XGL runtime library, the software pipeline, and the stroke font files.

```
% bin/setup_links
```

6. **Set the required environment variables to point to the appropriate locations, as follows:**
   - Set the `$XGLHOME` environment variable to point to the `<DDK_DIR>` directory.
   - Set the `$LD_LIBRARY_PATH` environment variable to point to the location of the XGL library and the OpenWindows libraries.

```
% setenv XGLHOME <DDK_DIR>
% setenv LD_LIBRARY_PATH $XGLHOME/lib:$OPENWINHOME/lib
```

7. **In the file `mk_cc_defs.include`, edit the lines CC5 and CCC5 for the appropriate hardware architecture so that the macros point to the location of the ANSI C compiler and C++ compiler on your system.**
   For example, on SPARC hardware running in the Solaris 2.x environment, you edit the lines as follows:

```
CC5-sparc = /opt/SUNWspro/SC2.0.1/acc
CCC5-sparc = /opt/SUNWspro/SC2.0.1/CC
```

8. **Change directory to the `src` directory.**

9. **Execute the `make opt` command to build the reference pipelines.**
   The Makefile builds the pipelines for the architecture you are running on and places the pipeline binaries in the `<DDK_DIR>/lib/pipelines` directory.

To build a single pipeline, execute the `make opt` command from the `<DDK_DIR>/src/dd/<pipeline>` directory. Note that the skeleton pipeline is intended as a template for your device pipeline; do not build the skeleton pipeline at this time. Note also that at compile time, the `cfb` pipeline requires several memory pipeline object files; therefore, you need to build the memory pipeline before building the `cfb` pipeline.

If no “Fatal Error” messages are displayed, the pipeline builds are complete.

2

Implementing the Skeleton Pixel-Level Graphics Handler 3

This chapter describes how to modify the skeleton source files provided with the XGL DDK product to create a pixel-level XGL graphics handler for your hardware device.

About Pixel-Level Rendering

The XGL graphics porting interface (GPI) consists of three layers of device pipeline interfaces, called LI-1, LI-2, and LI-3. The pixel layer of the XGL GPI is the lowest layer of the device pipeline. This layer, LI-3, is responsible for rendering pixels, vectors, and spans, but it leaves geometry processing and scan conversion to the XGL software pipeline. All device pipelines must implement routines for the LI-3 layer, since this layer is not implemented in the software pipeline.

To facilitate LI-3 implementation, the XGL GPI provides a utility object called RefDpCtx. The RefDpCtx object contains non-optimized implementations of all the LI-3 routines and several LI-1 routines. RefDpCtx writes to the hardware via one or more pixel objects called PixRects. A PixRect object is an abstraction of a buffer managed by the device, for example, the image buffer, Z-buffer, or accumulation buffer. PixRects represent the frame buffer pixel values to RefDpCtx. The PixRect object has methods to read and write pixels to the device, and RefDpCtx uses these methods to set pixel values on the device.

Your pipeline needs PixRects for the image buffer for a 2D pipeline, and for the image buffer, Z-buffer, and accumulation buffer for a 3D pipeline. The type of PixRect that you use to represent a particular buffer reflects your hardware. In

order to implement your graphics handler, you need to know the characteristics of your hardware. In particular, before you can write an LI-3 implementation with RefDpCtx, you must determine whether your hardware is memory-mapped and whether the Z-buffer and accumulation buffer are supported in hardware or handled in software.

When a pixel-level graphics handler uses RefDpCtx for rendering, all geometry and scan conversion functions are performed in the XGL software pipeline. The software pipeline returns LI-3 vector and span data, which the RefDpCtx object converts to pixel values. The device pipeline only accesses the hardware to read and write single pixel values.

Implementing LI-3 using RefDpCtx is a simple, quick way to port XGL to your hardware. Although rendering is slow, RefDpCtx provides complete coverage of functionality, as it supports texture mapping, blending, and transparency. Your XGL graphics handler will probably accelerate some primitives at the LI-1 and LI-2 layers, but implementing the LI-3 layer through the RefDpCtx object is an easy way to begin porting XGL to your hardware.

Steps for Implementing Pixel-Level Rendering

To implement LI-3 rendering with the skeleton files, follow these general steps:

- Choose a name for your graphics handler.
- Copy the skeleton files and rename them to the name of your pipeline.
- Edit the `Makefile` to rename the pipeline binary.
- Edit several skeleton device pipeline interface files to correspond to the capabilities of your device.
- Set hardware addresses to implement PixRect support.
- Build the pipeline.
- Test the pipeline.

The remainder of this chapter shows you how to complete these tasks.

▼ Choosing a Name for the Graphics Handler

First, choose a name for your graphics handler. A XGL graphics handler must be named according to the following convention:

```
xgl<COMPANY NAME><device name>.so.<major version>
```

where:

- <*COMPANY NAME*> is a 4-letter capitalized abbreviation for the company that implements the device pipeline.
- <*device name*> is the abbreviated name of the device, which should be an abbreviated form of the name of the corresponding kernel device driver located in the `/dev` directory.
- <*major version*> is the major release number of the DDK associated with the particular release of XGL that is compatible with this device pipeline.

For example, a Sun Microsystems Cg6 device pipeline for version 4 of the XGL GPI is named `xglSUNWcg6.so.4`, where SUNW is the company symbol, cg6 is the device name, and 4 is the major version number. For your pipeline, you only need to choose a company name and device name. The `xgl` and version number portions of the graphics handler name are added automatically by the Makefile.

The XGL versioning scheme is implemented as part of the device-independent library and as part of the DDK. The DDK header file `xgli/DdkVersion.h` defines the version number, which contains major and minor parts. When the pipeline is compiled, the major and minor version numbers are stamped in the pipeline file. The skeleton pipeline file `DpLibSkeleton.cc` includes the `DdkVersion.h` header file, as should your pipeline. For more information on versioning rules, see the *XGL Device Pipeline Porting Guide*.

▼ Copying and Renaming the Skeleton Files

The skeleton files provide the derived classes you need for your pipeline. To use the skeleton files, follow these steps:

1. **Execute the** `convert_skeleton` **script located in** `<DDK_DIR>/xgl/bin`**.** The `convert_skeleton` script creates a new directory hierarchy for your pipeline files and names the directory with your device name. The script then copies the skeleton files from the `<DDK_DIR>/src/dd/skeleton` directory to the new directory and renames the files with your device name.

The `convert_skeleton` script also updates all function names and variables to use the device name. As an example, for a pipeline named My_pipeline, the command would be:

```
<DDK_DIR>/bin/convert_skeleton [My_pipeline]
```

2. **Change directory to your pipeline directory.**

3. **To check that your pipeline builds, execute the `make opt` command.**
   The `make opt` command creates the `objs` directory for the skeleton pipeline object files and compiles and links the pipeline.

If the new pipeline builds without errors, you are ready to begin modifying the skeleton files for your pipeline.

Note – In the remainder of this guide, *skeleton pipeline* refers to your pipeline.

▼ Editing Files to Rename the Pipeline Binary

You need to change the `SYMBOL` variable in several skeleton files to your company name.

1. **Edit the skeleton pipeline Makefile to change `SYMBOL` to your company symbol in upper case letters.**
   Edit the following line:

```
LIB_NAME = xglSYMBOLskeleton
```

2. **Change directory to the `include` directory.**

3. **Edit the `xgl_errors_Skeleton.po` file to change `SYMBOL` to your company symbol in upper case letters.**
   Edit the following lines:

```
domain  “xglSYMBOLskeleton”
msgid  “SYMBOLskeleton-1”
msgid  “SYMBOLskeleton-2”
```

4. **Execute the** `make extract` **command to convert the error file text to the error file binary.**

▼ Editing the Skeleton Pipeline Interface Files

In this step, you will modify the skeleton header and source files for the device pipeline interface objects to match the capabilities of your hardware. The device pipeline interface objects connect the device-independent code with the device pipeline renderers. To enable you to implement a pixel-level graphics handler quickly, the XGL DDK product has provided a set of partially implemented files for these objects. All you need to do to render pixels on your device is add device-specific information to the generic skeleton pipeline interface files and, if necessary, write several functions.

Table 3-1 shows which skeleton device pipeline files require modification and which are provided ready to use. For more information on the device pipeline interface objects, see the *XGL Device Pipeline Porting Guide*. Comments in the skeleton source files provide information on how the skeleton pipeline implements these objects.

Table 3-1 Skeleton Pipeline Interface Files Needing Modification

Object	Source Files	Need Changes?
XglDpLib	`DpLibSkeleton.h` `DpLibSkeleton.cc`	No
XglDpMgr	`DpMgrSkeleton.h` `DpMgrSkeleton.cc`	Yes
XglDpDev	`DpDevSkeleton.h` `DpDevSkeleton.cc`	Yes
XglDpCtx2d XglDpCtx3d	`DpCtx*Skeleton.h` `DpCtx*Skeleton.cc`	Not at this time. See Chapter 4.

To edit the device pipeline interface files, follow these steps:

▼ Step 1: Edit the DpMgrSkeleton header and source files.

You need to update the XglDpMgr header and source files to add code for hardware initialization and update the `inquire()` method.

1. **In the `DpMgrSkeleton.h` file, add any variables needed by hardware initialization routines.**
   For example, you might need a structure to represent your hardware. This structure could contain the base register address, screen dimensions, and other information relevant to the physical device.

2. **In the `DpMgrSkeleton.cc` constructor, set the Boolean values for `memoryMappedImageBuffer` and `memoryMappedZBuffer` to `TRUE` if the image buffer or Z buffer are memory mapped.**

3. **In the `DpMgrSkeleton.cc` constructor, set the Boolean values `hwZBuffer` and `hwAccumBuffer` to `TRUE` if your hardware has a hardware Z buffer or hardware accumulation buffer.**
   The value of these variables are tested in the DpDevSkeleton object; they determine what type of PixRects are instantiated for your pipeline.

Tip ➨ If your hardware has a hardware Z buffer, but you want to get your port working quickly, leave the `hwZBuffer` variable set to `FALSE` to use the software pipeline Z buffer.

4. **In the `DpMgrSkeleton.cc` constructor, insert hardware initialization code.**
   The XglDpMgr class constructor is called once for each instance of your hardware device, so this is a convenient place to map in registers or frame buffer memory. For an example of hardware initialization for a GX device, which has a memory mapped image buffer and uses a software Z buffer, see the reference pipeline code for the GX frame buffer or see Appendix A, “Example Hardware Initialization Code” on page 43.

5. **In the `DpMgrSkeleton.cc` destructor, insert code to free any allocated resources.**

6. **Update the values in the `DpMgrSkeleton.cc` method `inquire()` to match the attributes of a window on your device.**
   The `inquire()` routine returns information on the acceleration features underlying a window and corresponds to the XGL API function `xgl_inquire()`. Applications use the information returned by `xgl_inquire()` to determine what is accelerated on the device they are running on.

   It is important to note that although `inquire()` appears to hold information about the frame buffer as a whole, the application that inquires about the device is actually requesting information on the window it is running in. For example, if your device accelerates more than one color type or provides double buffering on only one window at a time, your `inquire()` routine will need to determine what acceleration features were provided in a particular window in order to return accurate information to the application.

   To implement the `inquire()` method, do the following:

   - Set the value of the `name` variable to the name or symbol for your company. For example, the company symbol for Sun is SUNW, and the name for the GX device is cg6. Thus, on a GX device, the `xgl_inquire()` function returns `inq_info->name = SUNW:cg6`.
   - Update other values in the `inquire()` routine to match your hardware.

Tip ➨ You may want to implement acceleration on your device before updating the values in `inquire()`. Be sure to update this routine; one way that applications know how to use your hardware is by checking the values returned in `xgl_inquire()`.

▼ Step 2: Edit the XglDpDev source file.

You need to override device specific functions provided in the DpDev source file.

1. **Override the optional methods in `DpDevSkeleton.cc` to correspond to the functionality of your device.**
   `DpDevSkeleton.cc` provides virtual functions for the set of optional methods inherited from the DpDev class hierarchy. Override these methods if the default behaviors or returned values do not match those of your device. For example, if your hardware handles double buffering, you have

to override the DpDev virtual functions `setBuffersRequested()`, `setBufDisplay()`, and `setBufDraw()` to do what is appropriate for your hardware.

Note that the DpDevSkeleton function `updateDev()` handles updating the hardware when the user switches rasters. `updateDev()` updates the device-specific attributes by getting the current values of the raster attributes from device-independent XGL and calling the functions that you have overridden.

2. **Insert code to free any allocated resources in the DpDevSkeleton destructor.**

▼ Implementing PixRect Support

The RefDpCtx utility object assumes that the pipeline is able to set individual pixel values to the hardware. The PixRect objects provide RefDpCtx with the base address of the hardware or software memory.

Setting up PixRect support in the skeleton pipeline has been designed so that you only have to:

- Set Boolean variables that indicate whether your hardware image buffer or Z buffer are memory mapped, and set additional Boolean variables that indicate whether your device has a hardware Z buffer or accumulation buffer. You should have already set these Boolean values in the `DpMgrSkeleton.cc` file.
- Set the hardware address for the image buffer, and, if applicable, for a hardware Z buffer or hardware accumulation buffer. Where you set these addresses depends on whether your hardware is memory mapped or not.

PixRect Support for a Memory-Mappable Device

PixRect support for memory-mapped devices is provided in the XglPixRect subclass XglPixRectMemAssigned. This class provides a method that creates a PixRect object on existing hardware memory.

In the DpMgrSkeleton.h file, a PixRect of type XglPixRectMemAssigned is already allocated to represent the entire frame buffer. This PixRect is used as a resouce for the window PixRect. It contains low-level information about the frame buffer address and size.

- **To associate the frame buffer PixRect with your device, edit the DpMgrSkeleton.cc constructor to set the** fb_address **variable to the base address of your frame buffer.**
  The DpMgrSkeleton class constructor uses Xlib calls to get the height and width of your frame buffer and the depth of the window. In DpMgrSkeleton.cc, the XglPixRectMemAssigned method reassign() initializes this PixRect with the correct information for your hardware.

PixRect Support for a Non-Memory-Mappable Device

If your device is not memory mappable, or if only one buffer is accessible at a time, you must subclass from PixRect.h and override methods to do what is needed to access the hardware. The skeleton pipeline provides a subclassed PixRect class in the files PixRectSkeleton.h and PixRectSkeleton.cc. You can use these files to implement your device-specific PixRect class.

To implement your device's version of the PixRect class, edit PixRectSkeleton.cc as follows:

1. **Override the** setValue() **method with code to write the value of a single pixel to the screen.**
2. **Override the** getValue() **method with code to return the value of a single pixel.**
3. **If both the image buffer and the Z buffer are non-memory-mapped, edit the** validateBuffer() **method to set the hardware registers to the appropriate buffer.**
   When both the image buffer and Z buffer are not memory-mapped, validateBuffer() determines whether the image buffer or Z buffer is the current buffer and sets the hardware registers to the appropriate buffer. The setValue() and getValue() methods then render to or read from the correct buffer.
4. **If your frame buffer is more than 32 bits deep, override the methods** getValueByPointer() **and** setValueByPointer()**.**

3

▼ Building the Device Pipeline

Now you can build your LI-3 pipeline using the Makefile provided with the skeleton pipeline. Your pipeline will render to your hardware at the LI-3 level using RefDpCtx and the PixRect `setValue()` and `getValue()` calls.

1. **To build an optimized pipeline, execute the** `make opt` **command.**
   The pipeline binary is located in the `lib/pipelines` directory.

   Note that an optimized pipeline cannot be debugged easily. To build a debuggable pipeline, execute the `make debug` command.

2. **If you modified the** `xgl_errors_Skeleton.po` **file to add device-specific error messages, execute the** `make extract` **command.**
   This command creates the `.mo` error file binary for internationalization of error messages.

▼ Testing the Device Pipeline

To test your pipeline, you can run the `install_check` program in the SDK `demo` directory or run any application program. The `install_check` program displays some information about the hardware. To run `install_check`, key in `$XGLHOME/demo/install_check`.

You can also run the Denizen Test Suite provided with the XGL DDK product. The Denizen Test Suite is a set of verification programs that enables you to test the accuracy of your implementation. Denizen is a set of shell scripts and C programs that uses the XGL library to render objects and evaluate results. It creates a log of events, errors, and failures that can be compared to logs provided by XGL.

The Denizen Test Suite is installed from the DDK CD. Its default installation location is `/opt/SUNWddk/ddk_2.5.1/xgl/src/test_suite/denizen`. This directory contains reference images used for comparison testing, documentation on the test programs, and the `run_denizen.sh` shell script that executes the Denizen test suite. The README file contains information on `run_denizen.sh`.

To run Denizen, follow these steps:

1. **Set the required environment variables as noted in the INSTRUCTIONS file in the** `denizen` **directory.**
   Be sure to set the `FB_NAME` environment variable to your pipeline name.

2. **Execute the** `run_denizen.sh` **script.**
   The `run_denizen.sh` script runs the entire set of Denizen tests. To run one or more test areas, you can execute `run_denizen.sh` <*test area*>.

For more detailed information on running Denizen and comparing test results, see the *XGL Test Suite User's Guide*.

3

Implementing Accelerated Primitives

4

You should now be able to render pixels to your hardware using the RefDpCtx utility object. If pixel rendering is working, you are ready to implement accelerated renderers on your device. This chapter discusses the implementation of accelerated primitives. It also presents information on design issues to consider when implementing your pipeline

About Rendering and Attribute Handling

Rendering and context state are handled by structures defined in the XglDpCtx classes and objects XglDpCtx2d and XglDpCtx3d. The XglDpCtx base class includes a dynamic array of function pointers to renderers and to functions that handle state setting. This array, called the `opsVec` array, is inherited by the device pipeline XglDpCtx2d and XglDpCtx3d objects. A portion of the default `opsVec` array in the base XglDpCtx3d class looks like this:

```
.....
opsVec[XGLI_LI1_MULTIMARKER] = XGLI_OPS(XglDpCtx3d::li1MultiMarker);
opsVec[XGLI_LI1_MULTIPOLYLINE] = XGLI_OPS(XglDpCtx3d::li1MultiPolyline);
.....
opsVec[XGLI_LI_OBJ_SET] = XGLI_OPS(XglDpCtx3d::objectSet);
opsVec[XGLI_LI_MSG_RCV] = XGLI_OPS(XglDpCtx3d::messageReceive);
```

When the device pipeline is instantiated, its XglDpCtx object contains a set of `opsVec` array pointers specific to the device. In its version of the array, the device pipeline overrides some or all of the LI-1 and LI-2 entries to install

function pointers to its own accelerated renderers. Renderers that are not overriden remain set to the default software pipeline renderers. The device pipeline must fill in pointers to LI-3 functions, since these routines are device dependent and not provided by XGL.

At rendering time, the application primitive call is routed to the device pipeline by the XGL device-independent code. The device-independent code maps the C primitive call to a C++ internal call, and forwards the call and the application data to the device pipeline by calling the device pipeline `opsVec` entry for the primitive. If the device pipeline has installed a function pointer to one of its own renderers in the `opsVec` array, its renderer is called.

An example of a device `opsVec` array is provided in `DpCtx3dSkeleton.cc`. In this file, the skeleton pipeline installs function pointers for the LI-1 and LI-3 required functions. The skeleton pipeline also includes function pointers for the required attribute handlers. For all other LI-1 and LI-2 renderers, the skeleton pipeline inherits the default array entries that point to software pipeline routines. A portion of the skeleton `opsVec` array is listed below.

```
// LI-1 Raster operations
opsVec[XGLI_LI1_NEW_FRAME] = XGLI_OPS(XglDpCtx3dSkeleton::li1NewFrame);
opsVec[XGLI_LI1_FLUSH] = XGLI_OPS(XglDpCtx3dSkeleton::li1Flush);
opsVec[XGLI_LI1_COPY_BUFFER]
=XGLI_OPS(XglDpCtx3dSkeleton::li1CopyBuffer);
.....
// LI3 Primitives - Required
opsVec[XGLI_LI3_MULTIDOT] = XGLI_OPS(XglDpCtx3dSkeleton::li3MultiDot);
opsVec[XGLI_LI3_VECTOR] = XGLI_OPS(XglDpCtx3dSkeleton::li3Vector);
opsVec[XGLI_LI3_MULTISPAN] = XGLI_OPS(XglDpCtx3dSkeleton::li3MultiSpan);
.....
// Attribute handlers - Required
opsVec[XGLI_LI_OBJ_SET] = XGLI_OPS(XglDpCtx3dSkeleton::objectSet);
opsVec[XGLI_LI_MSG_RCV] = XGLI_OPS(XglDpCtx3dSkeleton::receiveMessage);
```

The device pipeline implementation of the attribute handling routines `objectSet()` and `messageReceive()` focuses on attributes that the pipeline is concerned with. The `objectSet()` routine handles attribute changes resulting from an application call to `xgl_object_set()`. The `messageReceive()` function handles attribute changes resulting from changes to XGL objects. Information on attribute changes is noted by the device-independent code and passed directly to the device pipeline through the `opsVec` array. See the skeleton pipeline `DpCtx2dSkeleton.cc` and `DpCtx3dSkeleton.cc` files for examples of these routines.

The opsVec array is designed to minimize overhead in the device-independent code during a primitive call. The device pipeline is notified immediately of a primitive call or attribute changes.

Rendering in the Skeleton Pipeline

The skeleton graphics handler provides example accelerated pipelines for 3D LI-1 multipolylines and for 2D LI-2 polygons. The renderers reside in the files Skeleton2dLi2Pgon.cc and Skeleton3dLi1Mpl.cc. Each skeleton pipeline renderer is designed as a set that includes these routines:

- A generic routine that evaluates incoming data and determines whether acceleration is possible
- A fast renderer that does a quick test of the application arguments and then sends application point data to the hardware

Each generic routine does the following:

- Checks the application data and handles other changes that the pipeline needs to be aware of, such as context changes.
- Determines the current attribute settings for the attributes that the pipeline is concerned with.
- Sets the opsVec[] entry to the default software pipeline routine or to the fast renderer, depending on the attribute settings. In some cases, the routine calls the software pipeline directly.

The device pipeline can set opsVec entries at object creation or at any time during program execution. Installing opsVec array entries during program execution is usually a result of attribute changes and can be done from the objectSet() routine. As an example, the skeleton pipeline sets opsVec array entries by doing the following:

1. At device pipeline initialization, the skeleton pipeline initializes the opsVec array with static renderers for the LI-1 and LI-3 routines.
2. When a primitive call occurs, the skeleton pipeline generic renderer determines whether acceleration is possible. It checks the following:

- If the application data (the point type) cannot be accelerated, the software pipeline is called directly to do the operations at that loadable interface level, but the `opsVec` entry remains set to the generic pipeline. If the point type can be accelerated, the generic renderer continues by checking the current attribute settings.
- If the current attribute settings can be accelerated, the generic renderer sets the fast renderer in the `opsVec` array and then calls it. If the current attribute settings cannot be accelerated, the generic renderer sets the default software pipeline renderer in the `opsVec` array and then calls it.

3. When attributes change, if the skeleton pipeline is concerned with the changed attributes, the `objectSet()` routine sends the new values to the hardware and resets the generic renderer in the `opsVec` so that it will be called to re-evaluate whether acceleration is possible the next time a primitive is called.

In your implementation, you will set the `opsVec` array entries to your accelerated renderers. For some primitives, you may want to have a pair of renderers as shown in the skeleton pipeline. For other renderers, you may need a family of renderers to handle optimized cases of attributes or point types.

Steps for Implementing the Skeleton Renderers

To use the skeleton pipelines, you must add code to send data to your hardware. Depending on your hardware and implementation, you may also need to modify these routines in other ways so that they correspond to your hardware. In addition, you need to modify the DpCtxSkeleton files to set up attribute handling and rendering for your device.

The sections that follow describe the implementation of the skeleton renderers in detail. If you plan to implement accelerated rendering at the LI-1 layer, read the section on LI-1 multipolyline on page 35; if your hardware is suited for LI-2 acceleration, read the section on LI-2 polygon on page 29. How you modify the skeleton code will vary depending on your implementation.

▼ Implementing the 2D Polygon Renderer

The `Skeleton2dLi2Pgon.cc` file provides an example of a simple 2D polygon renderer implemented at the LI-2 layer. The file contains two routines:

- A generic renderer, `li2GeneralPolygon`, that determines whether acceleration is possible. `li2GeneralPolygon` is the entry point for the device pipeline polygon renderer at LI-2. A device pipeline for a device that implements polygon acceleration at the LI-2 level will provide this routine.
- A fast renderer, `li2FastGeneralPolygon`, that sends data to the hardware given certain point types and attribute values.

The following steps take you through the sections of these routines in detail and examine the kinds of modifications you might need to make to implement them for your hardware.

▼ Step 1: Modify the generic renderer.

The generic polygon renderer, `li2GeneralPolygon`, checks for point type, context, primitive, and attribute changes. If no changes have occurred, the routine sets the fast renderer in the `opsVec` array and calls the `opsVec[]` entry to send data to the hardware.

Note – At the LI-2 level, data is given to device pipelines under the control of the XglPrimData object. At LI-2, the software pipeline has performed LI-1 processing and stored the API data internally in XglLevel format. The LI-2 device pipeline must extract point and facet data from the XglLevel object. The skeleton pipeline provides an example of the use of the XglLevel methods. For more information on how the software pipeline stores data and on the methods the device pipeline can use to extract this data, see the *XGL Device Pipeline Porting Guide.*

The code for the initial section of `li2GeneralPolygon` is listed below. To implement `li2GeneralPolygon` for your hardware, you may need to modify one or more statements in the initial section of the routine. These changes are listed following the code, and the numbers to the left of the code sample correspond to the list items. Note that some of the actual code comments in `li2GeneralPolygon` have been omitted in this listing.

```
void XglDpCtx2dSkeleton::li2GeneralPolygon(XglPrimData* pd)
{
    XglLevel*           level     = pd->getCurrentLevelData();
    Xgli_point_list*    pt_list     = level->getPointLists();
    Xgli_facet_list*    facet_lists = level->getFacetList();

    // Check point list changes.
①→   if (!pt_list) {
       return;
    } else if (pt_list->pt_type != XGL_PT_FLAG_F2D ||
       (facet_lists && facet_lists->facet_type != XGL_FACET_NONE)) {
        swp->li2GeneralPolygon(pd);
        return;
    }

    // Check for context change.
②→  if (dpMgr->lastDpCtx != this) {
       updateContext();
    }

     // Determine whether the primitive has changed.
③→   if (lastPrim != XGLI_LI2_POLYGON) {
       if (ctx->getSurfFrontCOlorSelector() ==
                        XGL_SURF_COLOR_CONTEXT) {
           // Set Context surface front color into hardware
           }
       lastPrim = XGLI_LI2_POLYGON;
    }

    // Lock window.
      WIN_LOCK(drawable) ;

    if (drawable->windowIsObscured()) {
        WIN_UNLOCK(drawable) ;
        return;  // Window is obscured; don't render
    }

④→  Xgl_boolean accelerate =
       (surfFrontFillStyle == XGL_SURF_FILL_SOLID)    &&

    // Pattern is dependent on style. This example only
    // accelerates style XGL_SURF_FILL_SOLID.
    // FpatPosition is dependent on pattern.
    // ctx->getSurfFrontFpat():
    // ctx->getSurfFrontFpatPosition():
```

```
(ctx->getRop()          == XGL_ROP_SRC)           &&
(ctx->getPlaneMask()    == (Xgl_usgn32)-1)        &&
(ctx->getSurfEdgeFlag() == FALSE)                 &&
(ctx->getSurfInteriorRule() == XGL_EVEN_ODD)      &&
(ctx->getSurfFrontColorSelector() == XGL_SURF_COLOR_CONTEXT) ;

// Antialiasing support on 2D Contexts is device dependent
// (ctx->getSurfAaBlendEq()     == XGL_BLEND_NONE)       &&
// (ctx->getSurfAaFilterShape() == XGL_FILTER_GAUSSIAN)  &&
// (ctx->getSurfAaFilterWidth() == 1)                    &&
```

1. Statement 1 in `li2GeneralPolygon` specifies which API data types are accelerated. The skeleton pipeline accelerates rendering if the point type is `XGL_PT_FLAG_F2D`. If a facet list is present, the facet type must be `XGL_FACET_NONE`. In any other case, the routine calls the software pipeline to perform LI-2 operations. The software pipeline may call back the device pipeline at the LI-3 level.

   For your implementation, consider what point types and facet types the hardware can accelerate. Then modify statement 1 as needed so that your pipeline corresponds with the capabilities of your hardware. For information on the complete set of XGL data types, see the *XGL Reference Manual*.

2. Statement 2 determines whether a context switch has occurred. If the XGL Context that the application is using has changed, the `updateContext()` routine in XglDpCtx2dSkeleton is called to update the view group interface object and the Context attributes.

   If your hardware has only one hardware context, use this statement to keep track of context switches. If your hardware has multiple hardware contexts, you may want to associate each hardware context with an XGL Context. In this case, you do not need to check for context changes. Note, however, that some applications may define many XGL Context objects, so you may want to monitor context changes even if your hardware has multiple contexts.

3. Statement 3 determines whether the primitive has changed. Because context state may be different for each primitive, you may want to check context state for some attributes. For example, you may want to get the Context color for the current primitive and set it on the hardware before rendering.

4. Statement 4 checks the current attribute settings to evaluate whether to use the skeleton pipeline accelerated renderer or fall back to the software pipeline. The skeleton pipeline accelerates rendering if the surface front fill style is solid, and the ROP, plane mask, surface edge flag, surface interior rule, and surface front color selector are set to the default values.

   Modify this section if you implement other values for these attributes. For example, if your hardware handles fill styles, your pipeline should check the fill attributes. If your hardware handles the ROP mode XOR, your renderer should check the ROP attributes.

The last section of `li2GeneralPolygon` sets the `opsVec` entry to `li2FastGeneralPolygon` if acceleration is possible, or calls the software pipeline if acceleration is not possible. The `opsVecDiDefault` routine reinstalls the software pipeline as the default `opsVec` entry. The `opsVec` entry remains set to the software pipeline or the fast renderer until an attribute changes. At that time, the `objectSet` routine sends the attribute changes to the hardware and reinstalls the generic `li2GeneralPolygon` in the `opsVec` array. After an attribute change, the next time rendering occurs, the generic renderer evaluates changes and determines which renderer to call. You do not need to modify this code for your device pipeline.

```
if (!accelerate) {
   // Unlock window.
   WIN_UNLOCK(drawable);

   // Set opsVec[] to default renderer.
   opsVec[lastPrim] = opsVecDiDefault[lastPrim];

   // Call renderer though opsVec[]
   (this->*((void(XglDpCtx2d::*)(XglPrimData*))
            (opsVec[lastPrim])
           )
   )(pd);

   return;
}

// Acceleration is possible. Set opsVec[] to fast renderer.
opsVec[lastPrim] =
   XGLI_OPS(XglDpCtx2dSkeleton::li2FastGeneralPolygon);

// Call renderer though opsVec[]
(this->*((void(XglDpCtx2dSkeleton::*)(XglPrimData*))
```

```
            (opsVec[lastPrim])
          )
    )(pd);

// Unlock window.
WIN_UNLOCK(drawable);
```

Note – In your implementation, you can call the software pipeline directly as `swp->li2GeneralPolygon(pd)` rather than through the `opsVec` array.

▼ Step 2: Modify the fast renderer.

The `li2FastGeneralPolygon` routine first tests to verify that nothing has changed. The test checks that the point list is valid, that the context and the facet type have not changed, and that the last primitive was sent from the `li2GeneralPolygon` group of renderers. If something has changed, the routine sets the generic renderer and calls it directly. Otherwise, it sends data to the hardware.

In your implementation, add code in the fast renderer to send data to the hardware. Note that the skeleton renderer does not handle surface edges. An example of how to access LI-2 edge flag data in a renderer that supports edges is provided in the `Skeleton2dLi2Pgon.cc` comments.

```
void XglDpCtx2dSkeleton::li2FastGeneralPolygon(XglPrimData* pd)
{
    XglLevel*           level    = pd->getCurrentLevelData();
    Xgli_point_list*    pt_list     = level->getPointLists();
    Xgli_facet_list*    facet_lists = level->getFacetList();

    Xgl_facet_type ftype = (facet_lists) ? facet_lists->facet_type
                                         : XGL_FACET_NONE;

    XglDrawable* localDrawable = this->drawable;
    WIN_LOCK(localDrawable);

   if (!pt_list | (int)dpMgr->lastDpCtx - (int)this |
          ftype - lastFacetTypeInfo.facet_type |
          XGLI_LI2_POLYGON - lastPrim) {

      WIN_UNLOCK(localDrawable);
```

```
        // Set default generic renderer, then execute.
        opsVec[XGLI_LI2_POLYGON] =
            XGLI_OPS(XglDpCtx2dSkeleton::li2GeneralPolygon);

         li2GeneralPolygon(pd);
         return;
    }

    Xgl_pt_i2d    *pts;
    Xgl_sgn32     num_pts;  // Number of points
    Xgl_usgn32    num_pl  = level->getNumPointLists();
    Xgl_sgn32     pt_size   = pt_list->geom_ptr.step_size;

    for (Xgl_usgn32 i = 0; i < num_pl; i++) {
        num_pts = pt_list[i].current_num_points;
        // Skip point lists with less than three points.
        if (num_pts < 3) {
            continue;
        }

        pts   = (Xgl_pt_i2d*)pt_list[i].geom_ptr.base_ptr;
        flags = (Xgl_usgn32*)pt_list[i].flag_ptr.base_ptr;

        //   my_hw = BEGINNING_OF_POLYGON
        for (Xgl_usgn32 j = 0; j < num_pts; j++) {

            //   Pass data to hardware.
            //   my_hw_x = pts->x;
            //   my_hw_y = pts->y;

            XGLI_INCR(flags, Xgl_usgn32*, flag_size);
            XGLI_INCR(pts, Xgl_pt_i2d*, pt_size);

        } // end for(j)
    } // end for(i)

    // Unlock window
    WIN_UNLOCK(localDrawable);
}
```

▼ Step 3: Update DpCtx2dSkeleton.

Before your implementation of 2D polygon rendering is complete, you must modify the `DpCtx2dSkeleton.cc` file as follows:

1. Modify the routine to add your renderer. You can uncomment the following line in `objectSet()`:

```
// opsVec[XGLI_LI2_POLYGON = XGLI_OPS(XglDpCtx2dSkeleton::li2GeneralPolygon);
```

2. Add code to the `objectSet()` routine to update your hardware context for all the XGL Context attributes that your pipeline is concerned with.

   The routine retrieves the current attribute values and sends them to the hardware. It also sets a flag, `li2_generalpolygon_change_renderer`, that indicates whether the generic renderer should be reinstalled.

When you have made these changes, your implementation of the skeleton 2D polygon renderers is complete. Use the Denizen test suite to test your implementation.

▼ Implementing the 3D Multipolyline Renderer

The `Skeleton3dLi1Mpl.cc` file provides an example of a simple 3D line renderer implemented at the LI-1 layer. The file contains three routines:

- A generic renderer, `li1MultiPolyline`, which determines whether acceleration is possible. `li1MultiPolyline` is the entry point for the device pipeline line renderer at LI-1. A device pipeline for a device that implements line acceleration at the LI-1 level will provide this routine.
- A fast renderer, `li1MplineFast`, that sends data to the hardware given certain point types and attribute values.
- A 3D model clipping routine, `li1MplineMC`, that model clips the data and calls the fast renderer to send the data to the hardware.

The following steps take you through the sections of these routines in detail and examine the kinds of modifications you might need to make to implement them for your hardware.

▼ Step 1: Modify the generic renderer.

The generic renderer, `li1MultiPolyline`, checks for point type, context, primitive, and attribute changes. If no changes have occurred, the routine sets the fast renderer in the `opsVec` array and calls the `opsVec[]` entry to send data to the hardware.

The code for the initial section of `li1MultiPolyline` is listed below. To implement this routine for your hardware, you may need to modify one or more statements in the initial section of the routine. These changes are listed following the code, and the numbers to the left of the code sample correspond to the list items. Note that some of the code comments in `li1MultiPolyline` have been omitted in this listing.

```
void XglDpCtx3dSkeleton::li1MultiPolyline(Xgl_bbox* api_bbox,
                                  Xgl_usgn32   api_num_plists,
                                  Xgl_pt_list* api_pt_list)
{
    // Determine whether API data can be accelerated.
    if (!api_pt_list) {
       return;
① →  } else if (api_pt_list->pt_type != XGL_PT_F3D) {
       swp->li1MultiPolyline(api_bbox, api_num_plists, api_pt_list);
         return;
    }

    // Create a local copy of the point type.
    Xgl_pt_type pt_type = api_pt_list->pt_type;

    // Determine whether the point type changed. If so,
    // update point type information.
    // The XgliUtPtTypeInfo utility saves information about
    // the point type that the fast renderer can use.
    if (lastPtTypeInfo.pt_type != pt_type) {
        XgliUtPtTypeInfo(pt_type, &lastPtTypeInfo);
       lastPtTypeInfo.pt_type = pt_type;
    }

    // Check for context change.
② →  if (dpMgr->lastDpCtx != this) {
       updateContext();
    }

    // Determine whether the primitive has changed.
③ →  if (lastPrim != XGLI_LI1_MULTIPOLYLINE) {
```

```
    if ((ctx->getCurrentStroke()->getColorSelector() ==
           XGL_LINE_COLOR_CONTEXT)) {
           // Set Context line color into hardware.
    }
    // Update other attributes that may have changed.
    lastPrim = XGLI_LI1_MULTIPOLYLINE;
 }

 // Lock window.
 WIN_LOCK(drawable) ;

 if (drawable->windowIsObscured()) {
     WIN_UNLOCK(drawable) ;
     return;  // Window is obscured.
 }

 // Determine whether the transform changed.
if (transformsChanged ||
    viewGrpItf->changedComposite(li1StrokeViewConcern)) {
    updateTransforms();
 }

 // Determine whether the window clip list changed.
 // If so, update the hardware clip list using the DpCtxSkeleton
 // sharedUpdateLi1Cliplist() routine.
 // Hardware clip list updating must happen while in WIN_LOCK().
 if (drawable->clipChanged()) {
    // Update window clip list related changes.
    sharedUpdateLi1Cliplist();
 }

④ → Xgl_boolean accelerate =
 (lineStyle == XGL_LINE_SOLID) &&
     (ctx->getRop()          == XGL_ROP_SRC)      &&
     (ctx->getPlaneMask()    == (Xgl_usgn32)-1)   &&
     (ctx->getDepthCueMode() == XGL_DEPTH_CUE_OFF) &&

     (cur_stroke->getAaBlendEq()     == XGL_BLEND_NONE)&&
     (cur_stroke->getAaFilterShape() == XGL_FILTER_GAUSSIAN)  &&
     (cur_stroke->getAaFilterWidth() == 1)        &&

     // getPattern():    Pattern is dependent on style. This
     //    example only accelerates style XGL_LINE_SOLID.
     // getAltColor():   AltColor is dependent on pattern.

     (cur_stroke->getCap()  == XGL_CAP_BUTT)      &&
```

```
    (cur_stroke->getJoin() == XGL_JOIN_DEVICE)    &&
   (cur_stroke->getColorInterp()   == FALSE)     &&
 (cur_stroke->getColorSelector() == XGL_LINE_COLOR_CONTEXT) &&
   (cur_stroke->getWidthScaleFactor() <= 1.0);
```

1. Statement 1 in `li1MultiPolyline` specifies which API data types are accelerated. The skeleton pipeline line renderer accelerates rendering if the point type is `XGL_PT_F3D`. For any other point type, the routine calls the software pipeline to perform LI-1 operations.

   For your implementation, consider what point types your hardware can accelerate. Then modify statement 1 as needed so that your pipeline corresponds with the capabilities of your hardware. For information on the complete set of XGL data types, see the *XGL Reference Manual*.

2. Statement 2 determines whether a context switch has occurred. If the XGL Context that the application is using has changed, the `updateContext()` routine in XglDpCtx3dSkeleton is called to update the view group interface object and the Context attributes.

   If your hardware has only one hardware context, use this statement to keep track of context switches. If your hardware has multiple hardware contexts, you may want to associate a hardware context with an XGL Context. In this case, you do not need to check for context changes. Note, however, that some applications may define many XGL Context objects, so you may want to monitor context changes even if your hardware has multiple hardware contexts.

3. Statement 3 determines whether the primitive has changed. Because context state may be different for each primitive, you may want to check context state for some attributes. For example, you may want to get the Context color for the current primitive.

4. The next section of `li1MultiPolyline` checks the current attribute settings as part of determining whether to use the skeleton pipeline accelerated renderer or fall back to the software pipeline. The skeleton pipeline accelerates rendering when the attributes are set to the default values.

   Modify this section if you implement other values for these attributes. For example, if your hardware handles line patterns, your pipeline will be concerned with the line pattern attributes.

The skeleton pipeline accelerates lines if the hardware has a hardware Z buffer. If Z buffering is enabled and the hardware does not have a hardware Z buffer, there is no acceleration.

```
if (ctx->getHlhsrMode() == XGL_HLHSR_Z_BUFFER &&
                           !dpMgr->hwZBuffer) {
     accelerate = FALSE;
 }
```

The last section of `li1MultiPolyline` sets the `opsVec` entry to the fast renderer, if acceleration is possible, or calls the software pipeline if acceleration is not possible. The routine also determines whether model clipping is enabled. If so, it sets the `opsVec` entry to the `li1MplineMC` routine, which uses the `XgliUtModelClipMpline` utility to model clip the application data and calls `li1MplineRenderer` to perform the rendering. If your hardware doesn't handle model clipping, you can use the `XgliUtModelClipMpline` utility to do model clipping in software. The utility returns model clipped data.

The `opsVec` entry remains set to the software pipeline or the fast renderer until an attribute changes. At that time, the `objectSet` routine sends the attribute changes to the hardware and reinstalls the generic renderer in the `opsVec` array. After an attribute change, the next time rendering occurs, the generic renderer again tests for changes, and determines which renderer to call.

```
if (!accelerate) {
   WIN_UNLOCK(drawable);

   // Set opsVec[] to default renderer.
   opsVec[lastPrim] = opsVecDiDefault[lastPrim];

   // Call renderer though opsVec[]
   (this->*((void(XglDpCtx3d::*)
       (SKELETON_PROTOTYPE_ARGS_MPLINE))
       (opsVec[lastPrim])
       )
   )(api_bbox, api_num_plists, api_pt_list);

   return;
}

// Acceleration is possible. Set opsVec[] to fast renderer.
li1MplineRenderer = XglDpCtx3dSkeleton::li1MplineFast;
```

```
// Model clipping?
if (ctx->getModelClipPlaneNum())
   opsVec[lastPrim] - XGLI_OPS(XglDpCtx3dSkeleton::li1MplineMC);
else
   opsVec[lastPrim] = XGLI_OPS(li1MplineRenderer);

// Call renderer though opsVec[]
(this->*((void(XglDpCtx3dSkeleton::*)
          (SKELETON_PROTOTYPE_ARGS_MPLINE))
          (opsVec[lastPrim])
        )
 )(api_bbox, api_num_plists, api_pt_list);

 WIN_UNLOCK(drawable) ;
}
```

Note – In your implementation, you can call the software pipeline directly as `swp->li1MultiPolyline()` rather than through the `opsVec` array.

▼ Step 2: Modify the fast renderer.

The `li1FastMultiPolyline` routine tests to verify that nothing has changed. The test determines whether the point list is valid, the context and the facet type have not changed, and the last primitive was from the `li1MultiPolyline` group of renderers. If something has changed, the routine sets the generic renderer and calls it directly. Otherwise, it sends data to the hardware.

In your implementation, add code in the fast renderer to send the application data to the hardware.

```
void XglDpCtx3dSkeleton::li1MplineFast (Xgl_bbox* api_bbox,
                    Xgl_usgn32   api_num_plists,
                    Xgl_pt_list* api_pt_list)
{
    XglDrawable* localDrawable = this->drawable;
    WIN_LOCK(localDrawable);

    if (!api_pt_list |
       (int)dpMgr->lastDpCtx - (int)this |
       api_pt_list->pt_type - lastPtTypeInfo.pt_type |
```

```
            transformsChanged |
            localDrawable->modifChanged() |
            XGLI_LI1_MULTIPOLYLINE - lastPrim) {

            // If acceleration not possible, unlock window
            WIN_UNLOCK(localDrawable);

            // Set default generic renderer (slower), then execute.
            opsVec[XGLI_LI1_MULTIPOLYLINE] =
               XGLI_OPS(XglDpCtx3dSkeleton::li1MultiPolyline);

             li1MultiPolyline(api_bbox, api_num_plists, api_pt_list);
             return;
        }

        float     *pts;
        Xgl_sgn32 num_pts;
        Xgl_sgn32 pt_size = lastPtTypeInfo.pt_size;

        for (Xgl_usgn32 i = 0; i < api_num_plists; i++) {
            num_pts = api_pt_list[i].num_pts;
            if (num_pts < 2) {
                continue;
            }

            pts = (float*)api_pt_list[i].pts.f3d;

            //  my_hw = BEGINNING_OF_MULTIPOLYLINE
            for (Xgl_usgn32 j = 0; j < num_pts; j++) {

            //  Pass data to hardware here.
                //  my_hw_x = pts[0];  // x
                //  my_hw_y = pts[1];  // y
                //  my_hw_z = pts[2];  // z

                XGLI_INCR(pts, float*, pt_size);

            } // end for(j)
        } // end for(i)

        WIN_UNLOCK(localDrawable);
    }
```

▼ Step 3: Update DpCtx3dSkeleton.

Before your implementation of 3D line rendering is complete, you must modify the `DpCtx3dSkeleton.cc` file as follows:

1. Modify the routine to add your renderer. You can simply uncomment the following line in the skeleton pipeline `objectSet()`:

```
// opsVec[XGLI_LI1_MULTIPOLYLINE =
            XGLI_OPS(XglDpCtx3dSkeleton::li1MultiPolyline);
```

2. Add code to the `objectSet()` routine to update your hardware context for all the XGL Context attributes that your pipeline is concerned with.

   The routine retrieves the current attribute values and sends them to the hardware. It also sets a flag, `li1_multipolyline_change_renderer`, that indicates whether the generic renderer should be reinstalled.

3. Add code to the `updateTransforms()` routine to send transform information to the hardware when transforms change.

4. Add code to `sharedUpdateLi1Cliplist()` in `DpCtxSkeleton.cc` to handle hardware clipping.

When you have made these changes, your implementation of the skeleton 3D polygon renderers is complete. Use the Denizen test suite to test your implementation.

Example Hardware Initialization Code

A

This appendix shows hardware initialization code for a GX frame buffer. The file consists of `include` files and the constructor for the XglDpMgr object for the GX frame buffer.

Note – If you are running on a GX frame buffer, and you built the GX reference pipeline `xglSUNWcg6.so.4` , and you want to build your pipeline using the GX intialization code listed below, you need to rename the `cg6` binary `xglSUNWcg6.so.4` and link your pipeline to the `cg6` name using the command `ln -s xglYourpipe.so.4 xglSUNWcg6.so.4`.

Hardware Initialization Code for the GX Frame Buffer

```
// ********************************************************
// *
// * Filename:DpMgrGXexample.cc
// *
// * Purpose:This is a C++ source file that contains a routine
// * used to initialize a physical hardware device managed by
// * the Dp manager and a routine that creates a Dp device object.
// *
// ********************************************************

#include “xgli/Drawable.h”
#include “xgli/SysState.h”// for XGLI_ERROR()
#include “DpMgrGXexample.h”
```

```
#include “DpDevGXexample.h”

// vvvvvvvvvvvvvvvvv GX example vvvvvvvvvvvvvvvvv
//
// Required package: SUNWDrvs
//
// for ioctl()
#include “/opt/SUNWddk/driver_dev/cgsix/sparc/cg6io.h”

#include <sysent.h>
#include <sys/mman.h>
#include <sys/cg6reg.h>
const int MegaByte = 0x100000;

// ^^^^^^^^^^^^^^^^^^ GX example ^^^^^^^^^^^^^^^^^^


class XglDpDev;
class XglRasterWin;


// *------------------------------------------------------
// *
// * XglDpMgrGXexample::XglDpMgrGXexample
// *
// * The constructor for the Dp manager object is called when
// * a new DpMgr object is created by the DpLib. This is done
// * once per physical device that is accessed.
// * Use this routine to initialize your hardware and to store
// * device-specific information.
// *
// *------------------------------------------------------


XglDpMgrGXexample::XglDpMgrGXexample(XglDrawable* drawable)
{
    creationOk = TRUE; // flag indicating successful hardware
                       // initialization. Default value is TRUE

    //
    // File descriptor or device name for opening your device.
    //

    int         fd   = drawable->getDevFd();
    const char* name =  drawable->getDeviceName();
```

```
    //=====================================================
    //=====================================================
    //  EDIT HERE:

// vvvvvvvvvvvvvvvvv GX example vvvvvvvvvvvvvvvvv
// Requires independent address space
    memoryMappedImageBuffer = TRUE;
// ^^^^^^^^^^^^^^^^^^ GX example ^^^^^^^^^^^^^^^^^^
   // Requires independent address space
    memoryMappedZBuffer     = FALSE;

    //  The value of the variables hwZBuffer and hwAccumBuffer
    //  are tested in DpDevGXexample. They determine which type
    //  of PixRect gets instantiated for your hardware. FALSE
    //  indicates that your hardware does not support Z buffering.

    hwZBuffer               = FALSE;

    //  Note: It is strongly recommended that if you have
    //  hardware accumulation, you implement li1Accumulate
    //  and li1ClearAccumulation. Otherwise, set
    //  hwAccumBuffer to FALSE. When this variable is
    //  FALSE, accumulation is done in software.See DpDevGXexample
    //  for allocation of the software accumulation buffer.

    hwAccumBuffer           = FALSE;

    // Insert hardware initialization here.

// vvvvvvvvvvvvvvvvv GX example vvvvvvvvvvvvvvvvv
    struct cg6_informgx_info;
    Xgl_sgn32 gx_pageoffset; // Offset of gx_physaddr from page bdry
    Xgl_sgn32 gx_physaddr;   // Adjusted gx_physaddr
    Xgl_usgn32gx_alloc_bytes;
    Xgl_usgn8*gx_base_addr;

    if(ioctl(fd, CG6IOGXINFO, &gx_info) == -1) {
       XGLI_DI_ERROR( (XglSysState*)NULL, “di-6”, XGL_OBJ,
                     NULL, NULL);
       creationOk = FALSE;
       return;
    }

    gx_alloc_bytes = gx_info.vmsize * MegaByte;
    if(gx_info.hdb_capable)
        gx_alloc_bytes *= 2;
```

```
   gx_pageoffset  = CG6_VADDR_COLOR & ( sysconf( _SC_PAGESIZE) - 1 );
    gx_alloc_bytes += gx_pageoffset;
    gx_physaddr    = CG6_VADDR_COLOR - gx_pageoffset;

    gx_base_addr = (Xgl_usgn8* ) mmap( (caddr_t)NULL,
              (size_t)gx_alloc_bytes,
              PROT_READ | PROT_WRITE, MAP_SHARED,
              fd, gx_physaddr );

    if(gx_base_addr == (Xgl_usgn8*)-1) {
       XGLI_DI_ERROR( (XglSysState*)NULL, "di-6", XGL_OBJ,
                     NULL, NULL);
       creationOk = FALSE;
       return;
    }
// ^^^^^^^^^^^^^^^^^^ GX example ^^^^^^^^^^^^^^^^^^
    //
    //  End hardware initialization here
    //=========================================================
    //=========================================================

    // Set this environment variable to test the error mechanism.
    // This example will abort this device pipeline if any
    // hardware initialization problems occur, output a device
    // specific error message, set creationOk to FALSE, and then
    // return immediately. In this case, the xpex pipeline
    // will be used.
    // Note: When this variable is set, your pipeline
    // will always abort.
    //
    char *hw_error = getenv("XGL_GXEXAMPLE_ERROR_TEST");
    if (hw_error) {
       XGLI_ERROR((XglSysState*)NULL,
              XGL_ERROR_NONRECOVERABLE,
              XGL_ERROR_RESOURCE,
              "SYMBOLgxexample-1",
              XGL_SYS_STATE,
              NULL, NULL);

       // Example of predefined error message (out of memory).
       // Notice _DI_
       XGLI_DI_ERROR((XglSysState*)NULL, "di-1",
                           XGL_SYS_STATE, NULL, NULL);
       creationOk = FALSE;
       return;
```

```
    }

    //
    // FOR MEMORY-MAPPED FRAME BUFFERS:
    // This section begins the initialization of the RefDpCtx
    // tilitisy for memory-mapped frame buffers. If your frame
    // buffer is not memory-mapped, you can ignore or delete
    // this section.
    //
    Xgl_usgn32linebytes;   // # bytes from one scan line to next
    Xgl_usgn32fb_width;    // Frame buffer width
    Xgl_usgn32fb_height;   // Frame buffer height


    //=========================================================
    //=========================================================
    // EDIT HERE:
    // If image buffer is memory mapped, set fb_address to
    // the base address of the image buffer; otherwise,
    // leave as NULL.
    // If z buffer is memory mapped, set z_buffer_address to
    // the base address of the z buffer; otherwise, leave as NULL.
    //
// vvvvvvvvvvvvvvvvv GX example vvvvvvvvvvvvvvvvv
    Xgl_usgn8*fb_address       = gx_base_addr;
// ^^^^^^^^^^^^^^^^^ GX example ^^^^^^^^^^^^^^^^^
    Xgl_usgn8*z_buffer_address = NULL;
    //=========================================================
    //=========================================================

    // X11 attributes
    Xgl_X_window        user_win;
    XWindowAttributes   wattrs;
    Window              window;
    Display*            dpy;
    intscreen;

    // Get the width and height of the frame buffer
    // and the depth of the window from the X server.
    //
    drawable->getDescriptor((void*)&user_win);
    dpy = (Display*)user_win.X_display;
    window = (Window)user_win.X_window;
    screen = (int)user_win.X_screen;
```

```
    // Identify properties of root window--our “frame buffer”.
    // Note that this example uses Xlib calls to retrieve information
    // about the hardware. You could also use an ioctl() for this.
    //
    XGetWindowAttributes(dpy, window, &wattrs);

    fb_width   = wattrs.screen->width;   // root window width
    fb_height  = wattrs.screen->height;  // root window height
    linebytes  = fb_width;

    //
    // If image and/or Z buffer is memory mapped, this initializes the
    // XglPixRectMemAssigned class. The PixRect object is
    // instantiated by the DpDevGXexample class.
    //
    if (memoryMappedImageBuffer)
        fbPixRect.reassign(fb_address, fb_width, fb_height,
                           wattrs.depth, linebytes);

    if (memoryMappedZBuffer)
        zPixRect.reassign(z_buffer_address, fb_width, fb_height,
                           wattrs.depth, linebytes);

    // End of section on initializing RefDpCtx for memory-
    // mapped frame buffers.
    // RefDpCtx initialization is continued in the DpDev object.

} // End of XglDpMgrGXexample::XglDpMgrGXexample
```