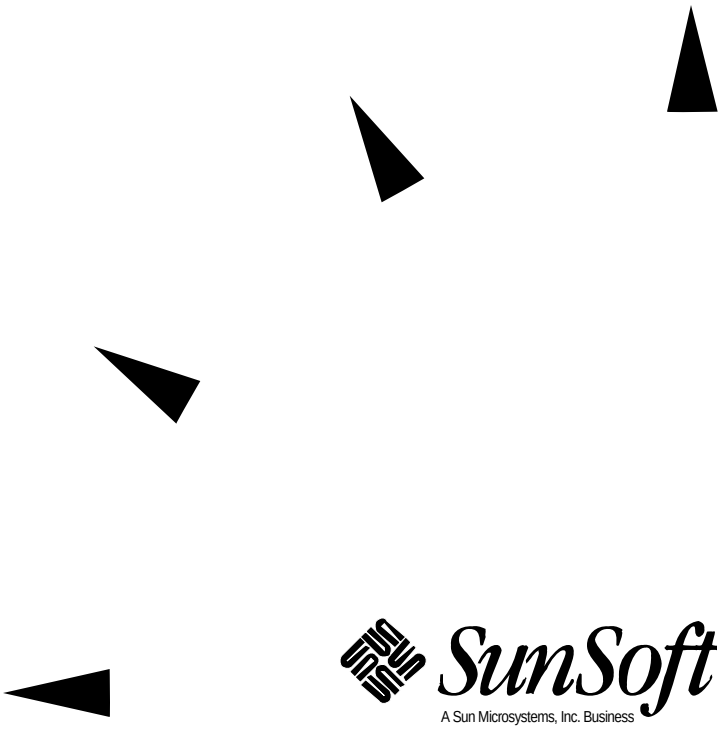
XGL Architecture Guide



2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Contents

Figures

Tables

Preface

The *XGL Architecture Guide* provides information on XGL™ architecture and presents details on the implementation of some of the key aspects of the architecture. This document also provides some information on the design of the loadable pipelines and describes XGL's object-oriented internal design and coding conventions. For information on writing a device pipeline, see the *XGL Device Pipeline Porting Guide*.

Who Should Use This Book

This document is designed for implementors of XGL device pipelines, and for XGL developers and maintainers.

Before You Read This Book

It is assumed that the reader is familiar with C++ and with the ideas of classes and class inheritance in C++.

How This Book Is Organized

This manual is organized into seven chapters:

Chapter 1, "Introduction to XGL,", provides a brief description of XGL functionality.

Chapter 2, "Overview of the XGL Architecture," introduces the XGL architecture.

Chapter 3, "XGL Class Structure," gives information on the device-independent and device-dependent class hierarchies.

Chapter 4, "Object Interactions," provides information on XGL object instantation and the process of pipeline creation.

Chapter 5, "Rendering and Handling State Changes," discusses the interactions between the device-independent internal code and the device pipelines.

Chapter 6, "Error Handling," provides information on the XGL error scheme.

Chapter 7, "XGL Coding Guidelines," presents the coding conventions used in XGL code.

Related Books

For information on writing a device driver for the XGL product, see the following documents:

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

For information on the XGL product, see the following documents:

- *XGL Programmer's Guide*
- *XGL Reference Manual*
- *Solaris XGL 3.0.1 Accelerator Guide for Reference Frame Buffers*

What Typographic Changes and Symbols Mean

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

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories	Edit your `.login` file. Use `ls -a` to list all files.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Introduction to XGL 1

Introduction to the XGL Product

XGL is a foundation library of two-dimensional (2D) and three-dimensional (3D) graphics functions designed to support a wide variety of geometry-based graphics applications. XGL provides direct mapping of graphics functionality to underlying hardware and implicitly uses hardware graphics acceleration whenever possible. Where hardware acceleration does not exist, XGL provides software emulation, allowing applications to run and produce nearly identical results on all platforms and graphics devices.

For application developers, the XGL library provides immediate-mode rendering and separate, complete 2D and 3D rendering pipelines. XGL has a rich set of graphics primitives as well as view and modeling transform support. Standard features include 2D and 3D primitive support (such as lines and polygons); depth cueing, lighting and shading; NURBS curve and surface support; and direct and indirect color model support. Advanced features include transparency, antialiasing, texture mapping, stereo, and accumulation buffer for motion blur and other special effects. Multi-primitive operators permit batching of simple primitive calls into groups. Besides providing a rich application programmer's interface (API) for application developers, the XGL library also serves as a foundation library for other graphics APIs, such as PHIGS, PEX, and GKS.

For developers of hardware graphics devices, the XGL product provides an open, well-defined graphics porting interface (GPI) in a loadable device pipeline architecture. This architecture enables hardware developers to build

display devices that optimize specific features of XGL. If a hardware vendor releases a new graphics device and provides an XGL device handler for that device, an application compiled for XGL will work without recompilation on the new device. By using the XGL library as a foundation-level interface, graphics devices under the Solaris™ environment will support any XGL application, and applications and graphics APIs written to the XGL library will run with any graphics device under the Solaris environment.

The XGL library resides directly above the hardware and firmware of the display device and graphics accelerator, minimizing software overhead within the XGL graphics pipeline. The relationship between XGL applications, high-level programming interfaces, and the XGL foundation library is illustrated in Figure 1-1.

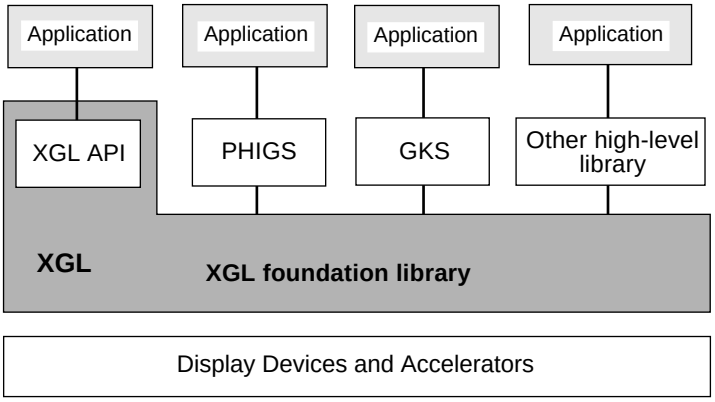


Figure 1-1 XGL API and Foundation Library

Running an Application in the X Window Environment

XGL runs within a window environment managed by an X11-R5 compatible server. XGL uses Sun's Direct Graphics Access (DGA) display technology, available in the OpenWindows environment, to accelerate XGL applications. DGA arbitrates access to the display screen between XGL and the X11 window system.

DGA technology enables XGL applications to achieve high-performance graphics when the application is running on the same machine as the X11 server and the hardware has DGA support. DGA synchronizes with the X11

server and allows XGL to send commands directly to the accelerator or frame buffer. This eliminates the overhead that results from building X11 protocol requests and passing them from the client to the server. An XGL program running locally with the server and using DGA is illustrated in Figure 1-2:

XGL Application and X11 Server on the Same Machine

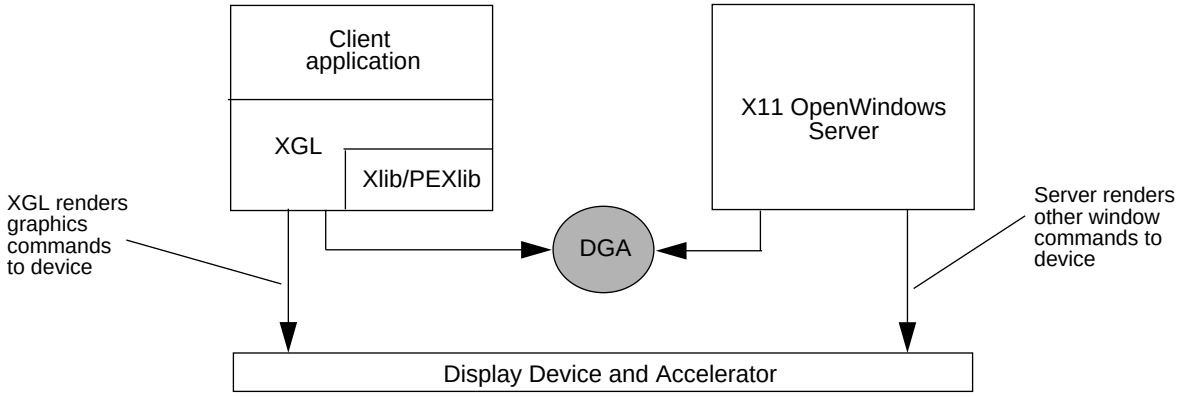


Figure 1-2 Using DGA in the OpenWindows Environment

When the XGL client program is running remotely, XGL uses Xlib or PEXlib to do all rendering. If the server includes the PEX extension and XGL has access to the PEX loadable library, XGL uses PEXlib to render. If PEX is not available, XGL uses Xlib for 2D rendering, and for primitives and rendering options that are not supported by Xlib, XGL does scan conversion to send the pixels through Xlib to the device. Figure 1-3 on page 4 llustrates remote rendering.

1

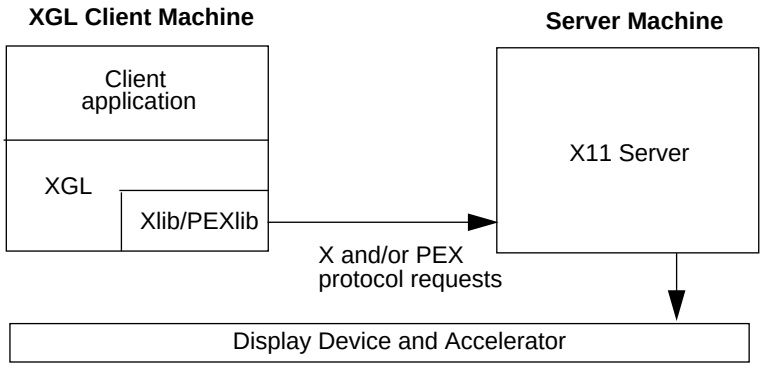


Figure 1-3 Using Xlib or PEXlib in the OpenWindows Environment

DGA ensures that XGL drawing operations are synchronized with the server. It also preserves the integrity of the display when windows are resized, moved, or obscured. DGA is transparent to the XGL application programmer. Applications or new graphics interfaces layered on top of XGL automatically benefit from features provided through XGL, such as DGA and remote rendering through X11 protocol.

Introduction to the XGL Programming Model

The XGL API is structured hierarchically through a series of abstract data types called *classes* (see Figure 1-4 on page 5). Applications can create *instances* of classes, where an instance of a class is an *object*. The class of an object determines the attributes it possesses and the operators that can act on it. XGL objects separate the application programmer from the specifics of the window system and the underlying hardware device. They present a consistent, device-independent graphics model that simplifies the work of the application programmer.

What Is an XGL Object?

An XGL object is an abstraction of a graphics resource. Each object describes a virtual component of the graphics rendering system and contains the information necessary to perform graphics operations, such as defining a line pattern or rendering a polygon. Display devices, for example, are known to the XGL programmer as Window Raster Device objects. Graphics state information

describing how XGL graphics primitives are drawn on the device is stored in a 2D Context or 3D Context object. Figure 1-4 illustrates the XGL class hierarchy from which graphics objects are derived.

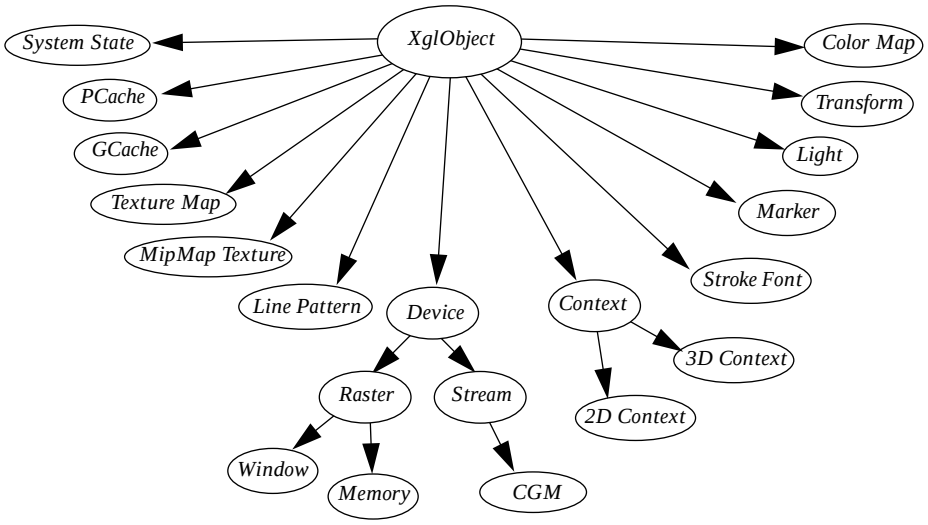


Figure 1-4 XGL Object Hierarchy

An XGL object is an instance of a class and contains state information (known as *attributes* in XGL) and member functions (known as *operators*) specific to its class. A class can be instantiated many times to create distinct objects that all belong to the class. Each object inherits its specific attributes and operators from its parent class. Operators perform predefined actions on objects of their parent classes.

How the XGL Programming Interface Works

An XGL application follows the general format of Xlib or X toolkit programming, using the event-driven model of X applications for interaction handling. With either Xlib or X toolkit calls, the application first creates the window system objects that it needs, such as a window for graphics display, and then opens the XGL system. When XGL is opened, it automatically creates

the System State object as well as internal objects that handle interactions between the device-dependent and the device-independent parts of the XGL system.

To set up a framework for rendering, the application must create a Device object, which is an abstraction representing the display device, and a Context object, which controls all rendering actions on a device. These objects are created using an XGL object creation operator that takes attributes describing the object as input arguments, creates an instance of the object, and returns a handle to the object. The application program must associate the Device object with the Context object before geometry can be rendered. When the Context object and Device object are associated, the application can use XGL primitives to do rendering and can pass control of the program to Xlib or an X toolkit to process events.

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

XGL Attributes

Attributes control much of the functionality of XGL, such as the appearance of rendered geometry, the way picking is handled, and the orientation of virtual device coordinate space. While some attributes are read-only, an application can change the value of most attributes at any time. Attributes are input as arguments to XGL operators, some of which take an attribute-value list as one of the input parameters.

XGL Operators

XGL provides a set of operators that applications can use to create objects and modify the attributes of objects. XGL includes several types of operators: generic operators used with all API objects, geometry-rendering operators, utility operators, and object-specific operators. The generic operators, listed in Table 1-1, provide a consistent method of working with all XGL objects.

Table 1-1 Generic XGL Operators

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

The rendering operators are the XGL drawing primitives. XGL provides a wide range of drawing primitives, including polylines, text, and filled areas such as rectangles, quadrilateral meshes, multiple single-boundary polygons, and multiple-boundary polygons. Primitives are available to an application as operators of the Context object.

Utility operators perform tasks such as clearing the Device viewport or copying blocks of pixels from a buffer in one Raster to a buffer in another Raster. Object-specific operators are provided for some objects; for example, the Transform object includes a set of operators that perform matrix operations.

Object Relationships

The relationship between XGL objects is managed internally by XGL object-management functions. When an application creates a new object, it can establish an association between the new object and an existing object using the `xgl_object_set()` operator, a connecting attribute, and the handles of the two objects. Table 1-2 on page 8 lists objects that are associated with the Context and Device objects as graphics resources.

Table 1-2 Object Relationships

User object	Related object
Raster Device	System State Color Map
Context	System State Device Transform Line Pattern Marker Pcache Stroke Font Light (3D Context only) Texture Map (3D Context only)

XGL Object-Oriented Programming

From the point of view of the architecture, XGL is an *object-oriented* system, since the internal code underlying the API adhers to the following basic principles of object-oriented programming:

- Objects consist of member data (attributes) and methods (operators).
- Objects derive from classes in a class hierarchy and inherit the data and functions of their parent classes.

To the application programmer, however, XGL is an *object-based* programming model, since an application can only define instances of classes and cannot create new classes to extend the class hierarchy. The application can access an object only via its object handle and operators; the object's actual data structures are not accessible to the application.

Overview of XGL Functionality

The XGL product is a software library that defines how graphics functions, such as transformations, lighting, shading, texture mapping, and other geometric operations, are performed. The library provides a full set of 2D and 3D primitives, which are listed in Table 1-3 on page 11. Broad coordinate-type support is provided for 2D and 3D pipelines, including:

- Integer, floating point, and double precision floating point types for 2D pipelines.
- Floating point and double precision floating point types for 3D pipelines.

Additionally, the XGL library provides features such as dynamic tessellation of NURBS curves and surfaces, surface trimming of NURBS surfaces, geometry caching, and backing store support.

The following sections provide a brief introduction to XGL functionality. For detailed information on the use of specific API object operators and attributes, see the *XGL Programmer's Guide* and the *XGL Reference Manual*.

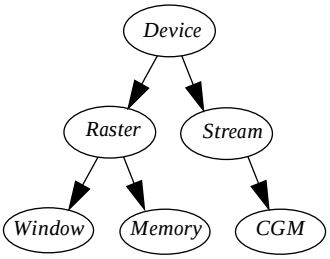
Opening and Closing the XGL Library

The XGL library is opened and closed by operators of the System State object. The System State object maintains information pertaining to all operations occurring during a single XGL session. Only one System State object can be created for any single XGL session.

System State attributes provide information on the location of font data and allow the application to set the default path used by XGL to access stroke font data files used in the text primitives. System State attributes also adjust the degree of error handling XGL performs and allow the application to supply its own error-handling functions in place of the XGL default error function.

Graphical Display

The XGL interface renders onto a graphical display through an XGL Device object and 2D or 3D Context objects. The display device is generally an X11 window linked to the XGL application through a window handle. The XGL application links the on-screen window to an XGL Raster Device object when the raster is created.



The Device object is an abstraction of a drawing surface. It provides a framework for interacting with hardware graphics devices in an abstract manner. The Device class itself is not instantiated to objects; instead, it is subclassed to raster devices, which represent two-dimensional rectangular arrays of pixels, and non-raster devices, which represent images that are not displayed via pixels. Raster device classes are instantiated to provide the following objects:

- Window Raster Device object – This object represents an area of the frame buffer that XGL will write into.
- Memory Raster Device object – This object represents a block of memory allocated from main memory. The pixels composing a Memory Raster object are in the application's memory space.

Non-raster objects are subclassed to provide Stream device objects. Stream devices provide a protocol-independent pipeline for creating formatted output such as Computer Graphics Metafile (CGM) output. The XGL Stream device maintains no protocol itself but relies on an instantiated third-party non-raster device pipeline to implement the appropriate output protocol data format. The XGl library includes CGM functionality that conforms to the ISO 8632:1985 standard.

Raster Attributes

Raster attributes allow an application to determine raster height, width, and depth. The Raster depth attribute defines the number of bits used to specify the color of one pixel in an XGL Raster. The depth of a Window Raster is device dependent. For a Memory Raster, three depths are supported: 1, 8, and 32. The 1-bit depth is used for creating stipple patterns.

The application can specify single or double buffering and can utilize hardware double buffering when it is available. If the hardware supports only a single buffer, color map double buffering can be performed. Backing store support can be enabled for a Window Raster; however, backing store and double buffering are not supported concurrently. The application can also set an attribute to specify a stereo window for rendering.

An application can set or get the starting address of the array of pixels set aside for an XGL Memory Raster, including the Memory Raster's Z-buffer. This allows the application programmer to read and write pixel data directly.

Primitives

The XGL library accepts a number of different representations of geometrical data, providing applications with flexibility in choosing an appropriate implementation. Data representation includes coordinate types and attributes associated with the rendered geometry. Most primitives can be rendered through both 2D and 3D pipelines. Available primitives are listed in Table 1-3.

Table 1-3 XGL Primitives

Operators	**Description**
`xgl_annotation_text()`	Renders annotation text (2D or 3D).
`xgl_multiarc()`	Draws a list of arcs (2D or 3D).
`xgl_multicircle()`	Draws a list of circles (2D or 3D).
`xgl_multi_elliptical_arc()`	Draws a list of elliptical arcs (3D only).
`xgl_multimarker()`	Draws markers at a list of points (2D or 3D).
`xgl_multipolyline()`	Draws a list of unconnected polylines (2D or 3D).
`xgl_multirectangle()`	Draws a list of rectangles (2D or 3D).
`xgl_multi_simple_polygon()`	Draws a list of polygons (2D or 3D).
`xgl_nurbs_curve()`	Draws non-uniform B-spline curves (2D or 3D).
`xgl_nurbs_surface()`	Draws non-uniform B-spline surfaces (3D only).
`xgl_polygon()`	Draws a single polygon (2D or 3D).
`xgl_quadrilateral_mesh()`	Draws a quadrilateral mesh (3D only).
`xgl_stroke_text()`	Renders stroke text (2D or 3D).
`xgl_triangle_list()`	Draws a triangle strip, triangle star, or batch or unconnected triangles (3D only).
`xgl_triangle_strip()`	Draws a triangle strip (3D only).

1

Graphical Context

The Context object is the central object that holds information about the rendering of geometric data. Context attributes control the position and appearance of geometry when it is rendered. Context operators include the drawing primitives and a number of utility functions. The utility operators include raster functions and pixel functions. The Context object can be associated with other objects that serve as graphics resources for the Context, such as the Line Pattern object and the Light object.

Rendering Attributes

Context graphics state attributes control many aspects of rendering, including the following:

- Line Characteristics – Context attributes set the polyline style to solid or patterned and define the color of a line. They also define the shape of the endpoints of lines and curves, set the line width scale factor, and define the shape of joins between line segments. An application can define a new line pattern by creating a Line Pattern object and attaching it to the Context object.
- Marker Characteristics – Context attributes define the size, color, and type of marker that is rendered. An application can define a new marker by creating a Marker object and attaching it to the Context object.
- Stroke Text Characteristics – Context text attributes define stroke text character height, width, spacing, and alignment. They also define text horizontal and vertical alignment, direction, and color. Multiple fonts can be attached to the Context object with Context attributes.
- Annotation Text Characteristics – Annotation text is text that is embedded in a plane parallel to the display surface. Context attributes define annotation text character height, character alignment, text alignment, and text direction.
- Transformation Characteristics – Context Transform attributes direct the conversion of geometric data from application coordinates to device coordinates. Transform objects are manipulated via Context object attributes.
- Surface Characteristics – Context attributes set the color used to fill surfaces and define the way in which surfaces are filled. They also enable hidden line and surface removal, control the drawing of silhouette edges around a

surface, and define how surface normals are calculated when they are not provided in application data. The application can define a fill pattern for a surface by creating a Memory Raster object and attaching it to the Context for the front fill pattern or the back fill pattern.

- Surface Edge Characteristics – Context attributes set the edge style, color, and width characteristics of the edges of surfaces (rectangles, circles, polygons, or arcs). An application can define a new pattern for a surface edge by creating a Line Pattern object and attaching it to the Context with a Context edge attribute.
- Surface Lighting Characteristics (3D attributes) – A 3D Context attribute sets the number of lights available to the Context; XGL then creates or destroys lights to match the specified number. Existing lights can be turned off or on, and they can be positioned and aimed to illuminate graphical objects for a desired effect. Context attributes control the global rendering properties of materials, such as material color, and the reflection coefficients for the various components of reflected light. An application can also create its own array of lights by creating a Light object and attaching it to the Context object.
- Surface Shading Characteristics (3D attributes) – A 3D Context attribute determines how a surface is shaded by defining the object's illumination style. The following illumination styles are available:
  - Objects can be drawn without lighting or color interpolation; in this case, the objects are drawn in their intrinsic color.
  - Objects can be drawn without lighting but with colors interpolated from the object vertex colors.
  - Objects can be drawn with lighting calculations performed at each facet; as a result, the entire object is drawn in the reflected color.
  - Objects can be drawn with lighting calculations performed at each vertex. If illumination per vertex is requested, XGL draws the object by linearly interpolating the reflected colors at the vertices across edges and subsequently across scan lines.
- Surface Transparency (3D attributes) – Context attributes enable applications to render transparent surfaces. XGL provides screen-door and blended (two-pass) transparency. If the transparency method is screen-door, a device-dependent *screen door* (a mesh that allows rendering of only some of the surface's pixels) is applied to the primitive. If the transparency method is blended, blending equations determine how the surface pixel values are blended with the background color or existing pixel values.

- Curved Surface Characteristics – Context attributes enable the application to control the smoothness of curved surface tessellation, to specify static or dynamic surface tessellation, and to define the placement of isoparametric curves on the surface. Surfaces can be trimmed using NURBS curves defined with Context curve attributes.
- Depth Cueing (3D attributes) – 3D Context attributes define the type of depth cueing rendered, specify the color that the primitive will be modulated to as the depth increases, and define how the depth cueing interpolation is calculated.
- Accumulation Buffers – Context attributes specify the destination buffer for an accumulation operation and specify the amount of jittering for the accumulation. When jittered images are averaged together, the resulting image is effectively antialiased.

Transformations

Geometric transformations specify a linear mapping from one coordinate space to another. An XGL Transform object represents a set of functions for transformation operations as well as a set of transformation matrices used to map geometry between coordinate systems on its way through the rendering pipeline. A transformation can be applied to the x, y, and z coordinates of geometric primitives using Transform objects. Data may be associated with a Transform object by loading a 4×4 matrix for a 3D transform or a 3×2 matrix for a 2D transform.

The Transform operators concatenate new matrices with existing Transforms and perform a variety of other functions, such as copy a Transform or multiply two Transforms. See the *XGL Programmer's Guide* or the *XGL Reference Manual* for more information on the Transform operators.

XGL Viewing Pipeline

The XGL viewing pipeline manages the transformation and rendering of geometric data from the application to the underlying display hardware. It moves geometric data along the viewing pipeline with a set of transformations and clipping attributes. During this process, lighting, depth cueing, shading, and transparency properties are applied. The XGL viewing model is based on the PHIGS model.

The Context object includes information that specifies the mapping and clipping of geometric data between coordinate spaces. Each Context has its own set of Transforms to implement the mapping between the sequence of coordinate systems. All Transforms default to the identify transform. To change a Transform, an application can use the Transform operators to build a new matrix, or it can input the matrix directly into the Transform object. When the new matrix is input to the Context as the target of a Transform operator, the Context object becomes aware of the change to its pipeline.

The application can update the following Transforms:

- Model Transform – The application controls the placement of each coordinate system in world coordinates with a transformation called the Model Transform. Although an application cannot access the Model Transform directly, it can change the Model Transform via attributes and transformations that set the *Local Model* Transform and the *Global Model* Transform. The Local Model Transform maps an element of the graphic scene to a global modeling space where the complete scene is assembled. The Global Model Transform maps the complete scene into world coordinate space. The Model Transform is the Local Model Transform preconcatenated with the Global Model Transform.
- Normal Transform (3D Contexts only) – 3D Context objects also have a *Normal* Transform, which maps normal vectors from model coordinates to world coordinates. The Normal Transform is the inverse of the Model Transform. XGL treats normal vectors as column vectors and points as row vectors.
- View Transform – The *View* Transform determines the application's viewing direction, orientation, and perspective, and maps coordinate values from world coordinates to virtual device coordinates. The transformation pipeline passes the geometric data assembled into world coordinates through the View Transform into virtual device coordinate space. The View Transform, view clip bounds, and VDC orientation collectively determine the generated image in VDC space.

XGL maps from Virtual Device Coordinate values to Device Coordinate values automatically using the VDC Transform. The VDC Transform performs this mapping by comparing the VDC and DC ranges. The VDC-to-DC mapping allows much of the pipeline from Model Coordinates to Virtual Device Coordinates to remain device independent. The VDC Transform is not accessible to the application.

Figure 1-5 illustrates the 2D viewing pipeline. Figure 1-6 illustrates the 3D viewing pipeline.

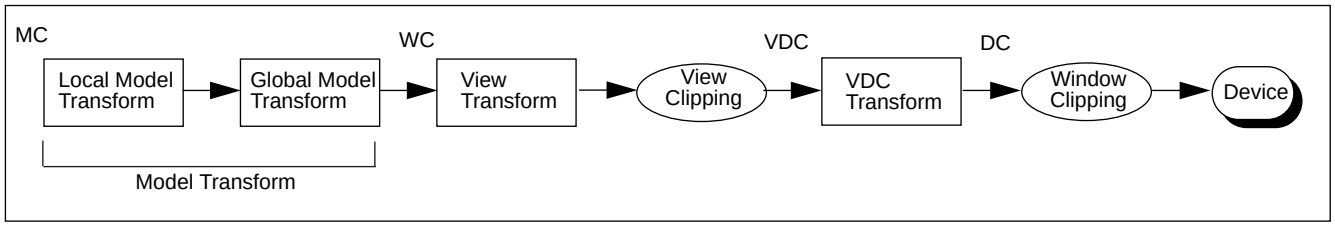


Figure 1-5 2D Viewing Pipeline

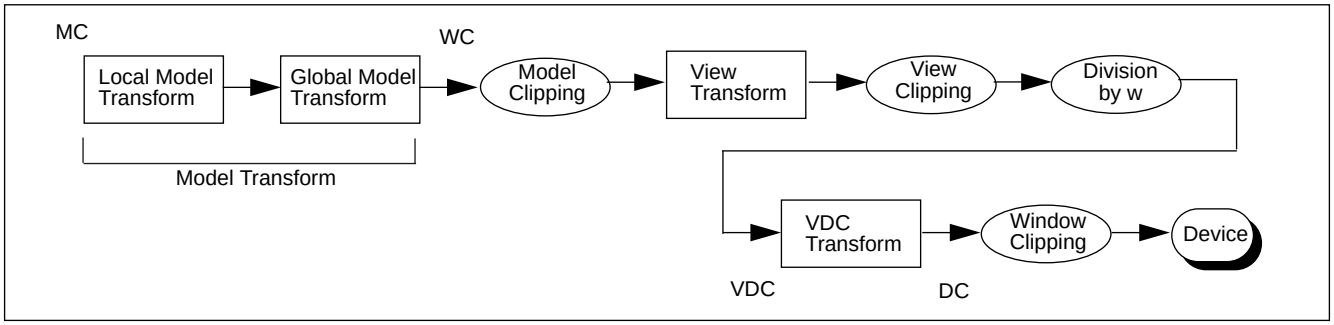


Figure 1-6 3D Viewing Pipeline

Color

The XGL color pipeline defines how colors are moved from the application's color type to the hardware device color type. The application's color type is defined with respect to the XGL Raster Device object associated with the Context. If the color type requested by the application is the same as the color type of the underlying hardware, no color conversion is necessary. The color values of the pixels are passed to the hardware without modification.

If the application color type is different from the hardware color type, a color conversion is required. In this case, the Color Map object is used to define color conversion from one color space to another. The application can perform color conversion in the following cases:

- The application can request indexed colors when the underlying hardware is true-color RGB. The indexed-to-RGB conversion is carried out using a color table stored in the XGL Color Map object.
- The application can request RGB colors when the underlying hardware is indexed. The RGB-to-indexed conversion is done using a color cube stored in an XGL Color Map object.

The Color Map object is an abstraction of a color table that is associated with a Device object. In simple cases with no conversion, the Color Map object is used as the software interface between the application programmer and the device color lookup table. The default color table, initialized at the creation of the Color Map object, is a two-element, black and white color table. When an indexed Window Raster Device object is created, either the default Color Map or an application-created Color Map is attached to the Raster.

Color Map Attributes

Color Map attributes fall into two categories: attributes used for indexed-to-RGB conversion, and attributes used for RGB-to-indexed conversion.

- For indexed-to-RGB conversion, Color Map attributes identify the color table structure, specify the number of entries in the color table, and determine the maximum number of entries the underlying hardware allows in a color table. When the color type is indexed color and vertex shading is required, color ramps must be defined in the color table for the colors used in shading. The application can write its own function to map colors from indexed to RGB color type.
- If an application's Device object color type is RGB and the underlying hardware is indexed, XGL uses a color cube and dithering to convert the application's RGB colors to the display's indexed colors. The color cube is located in the color table associated with an XGL Color Map. A Color Map attribute sets the size of the color cube, which is formatted as an array of three integers specifying the size of the red, green, and blue axes of the color cube.

Lighting and Shading

The XGL API supports flat shading and Gouraud shading. Light sources are defined as Light objects. A Light object can simulate ambient, directional, positional, and spot lighting, and it defines the illumination calculations that XGL uses for rendering. Lights are also used to compute the color of the vertices of the geometric data of a primitive.

Light object attributes specify light colors, positions, and directions, and determine the attenuation coefficients of the light. For a spotlight, attributes determine the angle of the beam and the light's attenuation characteristics.

Stroke Fonts

XGL provides a set of fonts that can be used for rendering text in an associated Context object. Text strings can be composed of characters from a single font or up to four fonts per Context. Text-encoding schemes permit multibyte text encoding and ISO encoding. Multibyte encoding (also known as Extended Unix Coding or EUC) uses control characters and bit patterns to specifiy different character sets within a character string. Allowing different encoding schemes and up to four fonts per Context supports internationalization extensions, allowing switching between different languages (character sets) within a text string. By default, a monospaced roman font is supplied with the Context object.

Text can be rendered in any orientation in 3D space using the `xgl_stroke_text()` operator, or it can be rendered parallel to the display surface using the `xgl_annotation_text()` operator. The appearance of rendered text is determined by Context stroke text and annotation text attributes.

Line Patterns

XGL provides a number of predefined line patterns that an application can request through the Context object. In addition, an application can create new line patterns using the Line Pattern object. Line patterns are used to define the pattern of lines, curves, and surface edges.

The default line style for rendering is a solid line. Context attributes change the line style to a patterned line and define the pattern as one of the predefined line patterns or as the pattern defined by the Line Pattern object. Context attributes also determine the color of the pattern, which can be drawn in a single color or in two alternating colors.

A new line pattern is specified by defining an array of alternating *on* and *off* segment lengths. The segments are used cyclically: when the end of a pattern is reached, the pattern starts over again from the beginning. For surface edges, the pattern wraps cyclically along edges and around corners until the end of the edge is reached.

Geometry Caching

XGL provides two buffering mechanisms to cache the geometric representation of data. One mechanism, the Pcache object, provides non-editable, non-hierarchical display list functionality for XGL. The Pcache object stores a sequence of primitives and relevant attributes for rendering at one time. Using the Pcache object, application programmers can write XGL code and tune it for the high performance that display lists can provide. The use of Pcache objects optimizes performance for most 3D graphics applications running on graphics adaptors with display list capabilities.

The second mechanism, the Gcache object, provides a facility to decompose complex primitives and accelerate their rendering through the use of simpler primitives that are suited for a particular graphics device. The Gcache object reduces the complexity of a primitive by allowing XGL to process the primitive into many simple primitives, storing the relevant attribute values. For example, a Gcache could contain the polylines that comprise a string of stroke text, or the triangles that make up a polygon. A Gcache also enables an application to decompose a primitive once and then render the result many times.

NURBS Curves and Surfaces

XGL includes advanced primitives for non-uniform rational B-spline (NURBS) curves and surfaces. NURBS are effective for representing simple or complex geometric shapes, and include powerful mathematical properties that provide ease of manipulation and use. XGL NURBS curves are general enough for users to generate uniform curves such as Bezier curves or uniform B-splines. NURBS attributes enable the application to control the precision of curve

rendering and the smoothness of the curve, and allow the application to balance the requirements of curve appearance and performance. XGL NURBS surfaces can be trimmed to define non-rectangular topologies, and the display of the surfaces can be enhanced with isoparametric curves.

The display of NURBS curves and surfaces can be improved using Gcache objects. Much of the processing time required for NURBS curve and surface display is absorbed during the creation of the Gcache object. This can substantially speed up rendering of NURBS curves and surfaces.

Texture Mapping

The XGL library supports 2D texturing of 3D surface primitives using a texturing raster image specified by the MipMap Texture object through the Texture Map object. Textures are defined in a normalized texture space (u,v), with the *u* and *v* coordinates defined in the range 0.0 to 1.0.

Mapping of the texture onto a polygon is accomplished by deriving (*u*,*v*) values from the vertex, normal, or data fields of the polygon vertices. For example, if a texture image is mapped completely onto a four-sided polygon (in a simple manner with no wrapping), the lower vertex of the polygon would be represented by the (u,v) coordinate pair (0.0,0.0), the upper left vertex by (0.0,1.0), the upper right by (1.0,1.0), and the lower right by (1.0,0.0).

The result of the texturing operation can be applied to different stages of the rendering pipeline. XGL supports sampling methods such as point, bilinear, and trilinear to obtain texture values and supports several color composition techniques, such as blend, decal, and modulate.

Note – At this release, the Texture Map object should be used for texture mapping. The Data Map Texture object is retained for backward compatibility, but it will be removed from the XGL library at a future release.

Overview of the XGL Architecture 2

This chapter presents an overview of the XGL architecture. It provides an introduction to the following topics:

- Goals of the architecture
- Overview of the device-independent and device-dependent components of the architecture
- Architecture of the device pipelines
- Role of the software pipeline in the rendering process
- Context state handling
- Window system and device pipeline interaction
- Color

More detailed information on these and other aspects of the XGL architecture is provided in later chapters of this manual.

The following terms and acronyms are used in this chapter and throughout the remainder of this book in discussions of the XGL architecture: *Core* and *DI* refer to the device-independent parts of XGL; *device pipeline* (*Dp*) refers to the device-dependent parts of XGL.

2

XGL Architecture Design Goals

The XGL architecture enables independent software vendors (ISVs) and independent hardware vendors (IHVs) to port XGL to a wide range of graphics applications and hardware platforms. The XGL architecture was designed to fulfill the following goals:

- Define a graphics porting interface (GPI) that enables structured porting of XGL to new display devices (frame buffers, accelerators, etc.).
- Clearly define the relationships between internal objects.
- Optimize XGL device-independent performance to ensure that graphics accelerators are able to achieve their maximum possible performance.

The XGL library does not contain any device-dependent code. It contains only device-independent code and utilities (such as lighting calculations). The device-dependent code, which controls graphics hardware, resides in separate files called device pipelines.

Basic Components of the XGL Architecture

At the most basic level, XGL has two components: the device-independent component and the device-dependent graphics pipelines. The device-independent code functions as the interface between the application program and the graphics pipelines. The pipelines turn geometric primitives and their state attributes into pixel data that is displayed on a graphics hardware device or written into memory. These basic components are illustrated in Figure 2-1:

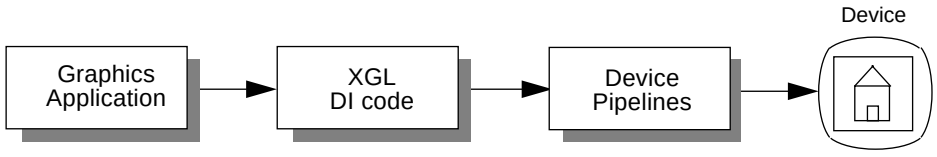


Figure 2-1 Basic View of XGL Architecture

Device-Independent Library

The device-independent component of the library contains class hierarchies for internal and API objects and abstract pipeline class interfaces to the underlying pipeline operations. It takes care of a variety of device-independent tasks, including:

- Mapping XGL C API function calls to internal C++ functions
- Creating and managing API graphics objects
- Providing utility functions to facilitate the implementation of device pipelines

Graphics Pipelines

The graphics pipelines include one or more device pipelines and a software pipeline. The device pipeline code resides in libraries that are dynamically loaded at runtime. When an XGL application begins execution, XGL determines what device the application is running on and loads the pipeline library needed to control that device. Thus, device support is not contained within XGL but is *loadable* at runtime.

The XGL graphics porting interface consists of three layers of pipeline interface, with each layer responsible for specific rendering functions. The top layer, 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 way that these layers are implemented differs between the device pipelines and the software pipeline.

Device Pipelines

Device pipelines written at the LI-1 layer typically implement the full graphics pipeline for each primitive. An LI-1 pipeline takes the points defining a primitive and transforms, lights (in the 3D case) and clips the geometry, performs scan conversion, and renders pixels on the device. Thus, an LI-1 pipeline will normally start the processing of geometry at the LI-1 layer and continue, including all the functionality below LI-1. Device pipelines written at

the second layer (LI-2) perform scan conversion and render pixels on the device. A device pipeline port to the lowest layer (LI-3) is responsible only for rendering pixels. IHVs can implement different GPI functions at different layers to tailor a port for a particular device.

Figure 2-2 illustrates the loadable interface layers.

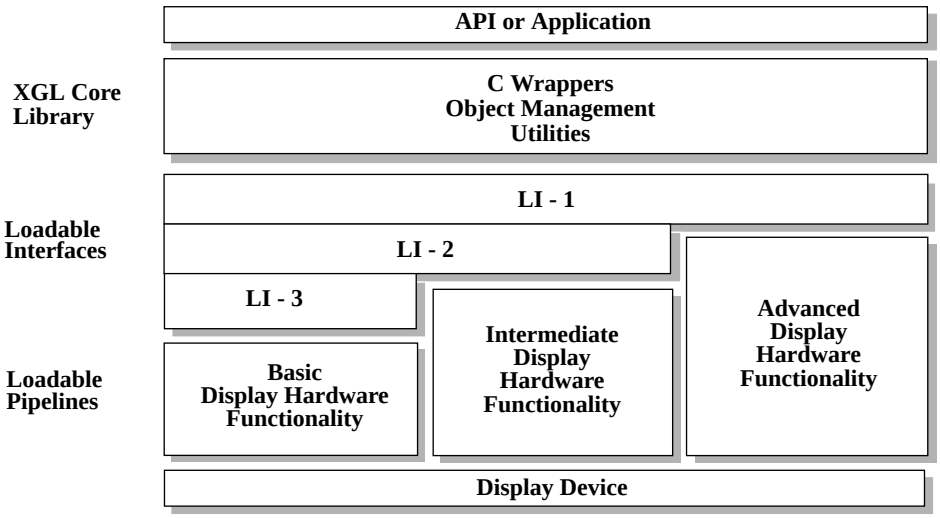


Figure 2-2 High-level View of the Loadable Interface Layers

XGL Software Pipeline

The XGL product provides a software implementation of most of the primitives in the LI-1 and LI-2 layers. This allows IHVs to port to the level of functionality appropriate to a device. IHVs can make full use of the capabilities of their device and use the XGL software pipeline for functionality that the device lacks.

The software pipeline consists of functions that implement the geometry pipeline and scan conversion functions in software. The functions in the top layer (LI-1) of the software pipeline take the points defining the primitive and transform, light (if necessary), and clip the geometry. The second layer (LI-2) is responsible for scan converting more complex primitives like polygons and

polylines. The software pipeline does not include LI-3 functions, since these are device dependent. Therefore, at a minimum, an IHV must provide a set of LI-3 functions for a device.

How the Device Pipeline and Software Pipeline Work Together

At rendering time, if the device pipeline has implemented an LI-1 function, the flow of control first goes to the device pipeline at the LI-1 level. The pipeline determines from the setting of API attributes whether it can or cannot render the primitive at that level. If it can render the primitive, it will generally perform all the operations necessary for rendering from the LI-1 level to the hardware. However, the architecture allows the flow of control to go to the device pipeline at LI-1 for part of the work and go to the software pipeline to complete LI-1 operations or to do LI-2 level processing. In general, when a device pipeline implements an LI-1 primitive, there is a match between the primitive-attribute combinations and what the accelerator can do.

As with LI-1, if the device pipeline receives a primitive at the LI-2 level, the flow of control will usually stay within the device pipeline until the primitive is rendered. A typical scenario for a device pipeline is that it will support some combinations of primitives and attributes at the LI-1 level, some at the LI-2 level, and some at the LI-3 level. For example, the pipeline might accelerate solid lines at LI-1 but support dashed lines at LI-2. To render dashed lines, the device pipeline would let the software pipeline handle the LI-1 layer geometry operations, and it would take over the rendering operations at the LI-2 layer.

By allowing the device to switch to the software pipeline, XGL provides a device pipeline with flexibility in how it renders a primitive. The pipeline can be written to fully accelerate a primitive from model coordinates to device coordinates, to partially accelerate a primitive and fall back on the software pipeline for some of the rendering tasks, or to not accelerate the primitive and let the software pipeline perform all but the most basic rendering tasks. Whenever a primitive or a rendering attribute for a primitive is not supported by the hardware, the device pipeline can fall back to the software pipeline for some or all of the processing for rendering. This dynamic decision making is one of the primary design features of the XGL architecture.

Writing a Device Pipeline

The task of writing device pipeline functions at different levels varies in complexity. Writing functions at the LI-1 layer is complex and may require a significant amount of time. Table 2-1 summarizes the functionality of a device pipeline port at a particular layer.

Table 2-1 Functionality of the Device Pipeline Layers

Layer	Responsibilities
LI-1	Must handle all aspects of processing an XGL primitive and all rendering operations, including scan conversion and pixel painting.
LI-2	Assumes responsibility for rendering but leaves geometry processing operations to the XGL software version of the LI-1 layer.
LI-3	Requires implementing span and dot renderers, but all other operations needed to process a primitive and reduce it to the pixel level are provided by XGL's default software implementation.

As mentioned above, a minimal port of XGL to a device must include functions at the LI-3 layer, since functions at this layer are not provided by the software pipeline. However, XGL provides a utility object called RefDpCtx (Reference Device Pipeline Context) that can help a pipeline writer quickly implement LI-3 functions. See the *XGL Device Pipeline Porting Guide* for information on RefDpCtx.

Figure 2-3 on page 27 illustrates the stages of the device and software pipelines as well as some of the components of the XGL core. For more information on the architecture of the device pipelines, turn to page 28.

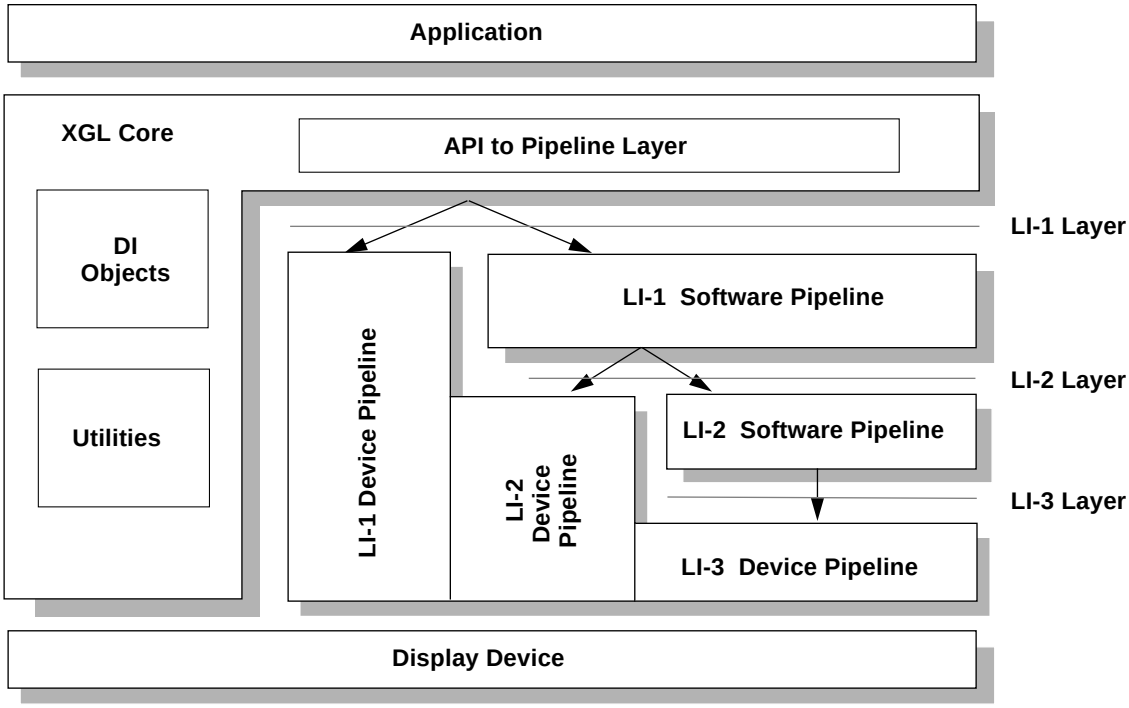


Figure 2-3 High-Level View of the XGL Architecture

Note – The XGL library provides a Stream device that supports output to a file; however, the bulk of this chapter assumes that you are writing to a pixel-based frame buffer.

2

Overview of the Device Pipeline Architecture

Because XGL runs in a workstation windowing environment, it must provide support for applications with multiple windows, for multiple applications per frame buffer, and for multiple physical display devices. It must also include support for a range of hardware devices. The XGL loadable pipeline architecture was developed to explicitly address the wide range of geometry-oriented graphics devices currently in use and to provide a foundation for future display technology.

To provide the support necessary for a variety of hardware devices, XGL includes a set of well-defined, abstract interfaces that link the device pipelines with the XGL core. The XGL graphics library does not contain any device-dependent code and hence cannot render directly to a frame buffer itself. To access a frame buffer, XGL loads the device pipeline appropriate for that frame buffer. Although the term *pipeline* traditionally refers to the steps in the rendering process, in XGL the design of the loadable pipelines includes components that serve as the framework connecting the device-independent code to the device pipeline code.

The device pipeline is made up of a number of C++ classes. The classes are abstract classes in that they define a set of functions that must be implemented by the device pipeline. As part of the process of writing a device pipeline, the implementor must subclass the XGL-provided abstract classes and implement them to provide device-specific classes and objects.

Architecture at the API Level

At the API level, XGL has two primary entities that are responsible for rendering: the Context object and the Device object. The Context object is the central object for the application program; it acts as a clearing house for attribute information and does the work of drawing graphics. Geometry and related attributes are rendered through the Context object to the physical device via the Device object. Multiple Context objects can be set up for each Device object, as would be the case if an application program wanted to render 2D and 3D geometry. Figure 2-4 shows a possible set of Context and Device objects that might be attached to one window of an application program. The Window Raster Device object represents application window 1.

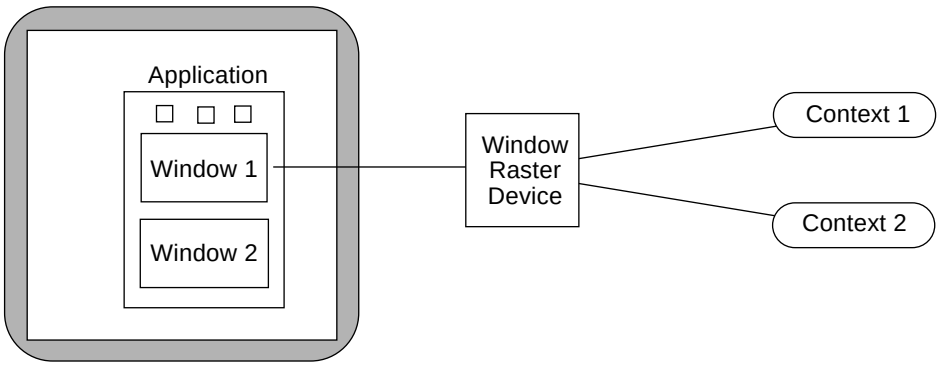


Figure 2-4 Architecture at the API level

Internal Pipeline Architecture

Internally, XGL uses a set of objects to connect the device-independent Context and Device objects with the device-specific pipeline code. These objects are:

- Device pipeline context object
- Device pipeline device object
- Device pipeline manager object
- Device pipeline library object

The sections that follow introduce and illustrate these objects.

Device Pipeline Context Object

The device pipeline-context (DpCtx) object is the device-dependent representation of a Context object for a specific hardware device. When the application program associates a Context object with a Device object, XGL creates a DpCtx object for the Device and Context pair.

The DpCtx object contains the actual rendering functions for a device and keeps track of Context state. Figure 2-5 shows the DpCtx objects that represent two Contexts associated with the Window Raster Device object. In this figure, the objects in the pipeline framework are shown as ovals with dark borders.

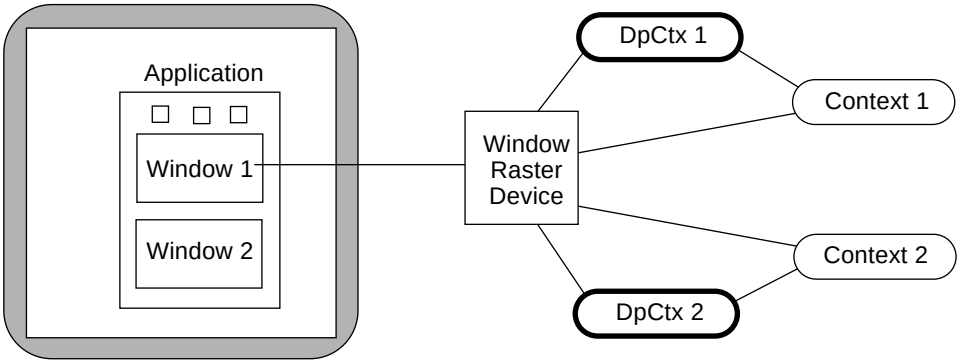


Figure 2-5 Device Pipeline Architecture: DpCtx Object

Device Pipeline Device Object

Figure 2-6 introduces the device pipeline device object (DpDev). This object manages the DpCtx objects corresponding to each Device object. The DpDev object is the device-dependent part of the XGL Device object, and it is used to manage the device dependencies of the DpCtx objects.

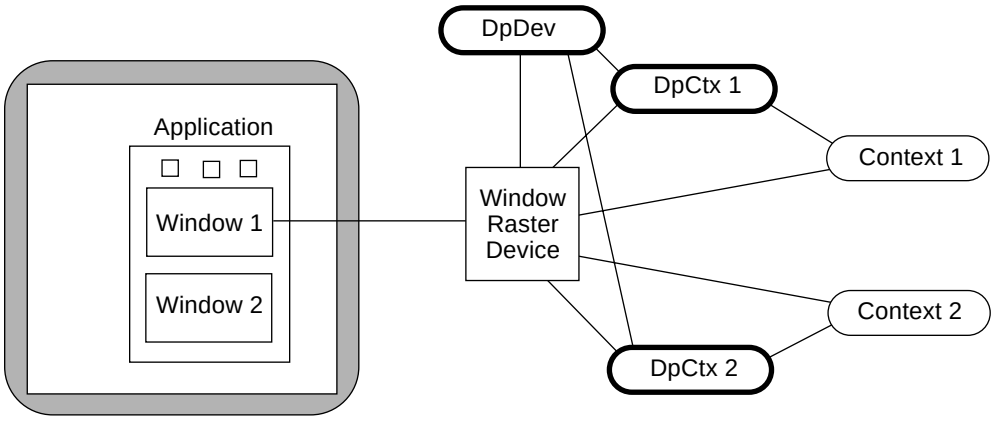


Figure 2-6 Device Pipeline Architecture: DpDev Object

Device Pipeline Manager Object

Figure 2-7 presents the device pipeline manager (DpMgr) object. This object handles various categories of devices, such as frame buffers. The DpMgr object is unique in that is does not have a corresponding API-visible object, whereas the DpDev object corresponds to the API Device object, and the DpCtx object corresponds to the associated API Device and Context objects.

The DpMgr object manages the DpDev objects for the device. If the category is a frame buffer, the DpMgr initializes the frame buffer. Figure 2-7 shows a single DpMgr object for the two DpDev objects that represent two application windows.

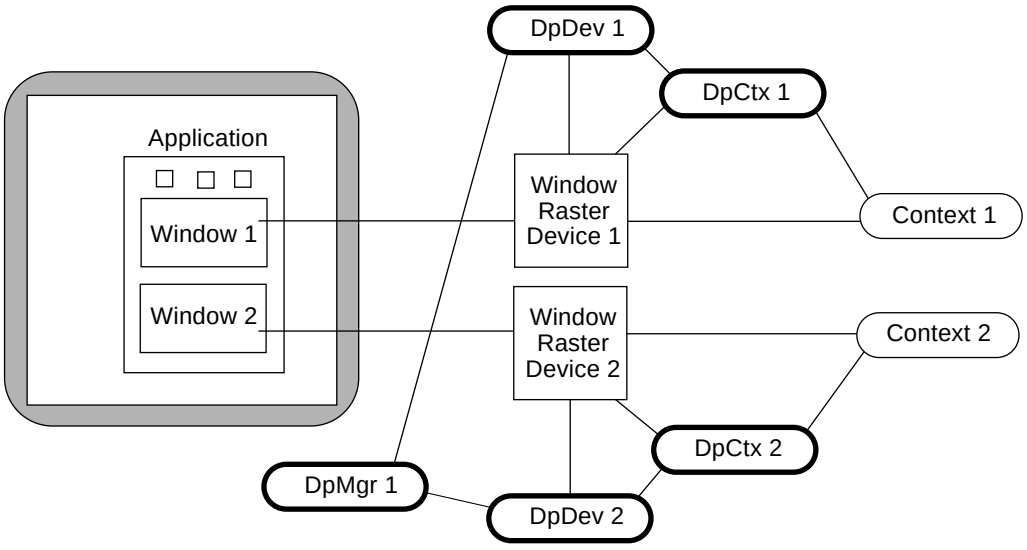


Figure 2-7 Device Pipeline Architecture: DpMgr Object

Device Pipeline Library Object

The device pipeline library (DpLib) object represents the shared library for the device pipeline. Figure 2-8 shows the DpLib object in a system with two frame buffers of the same type. In this diagram, a single application has opened two windows on one screen and one window on another screen. The DpLib object allows more than one DpMgr to share hardware and software resources. For more information on the pipeline framework, see the subsequent chapters of this book and the *XGL Device Pipeline Porting Guide.*

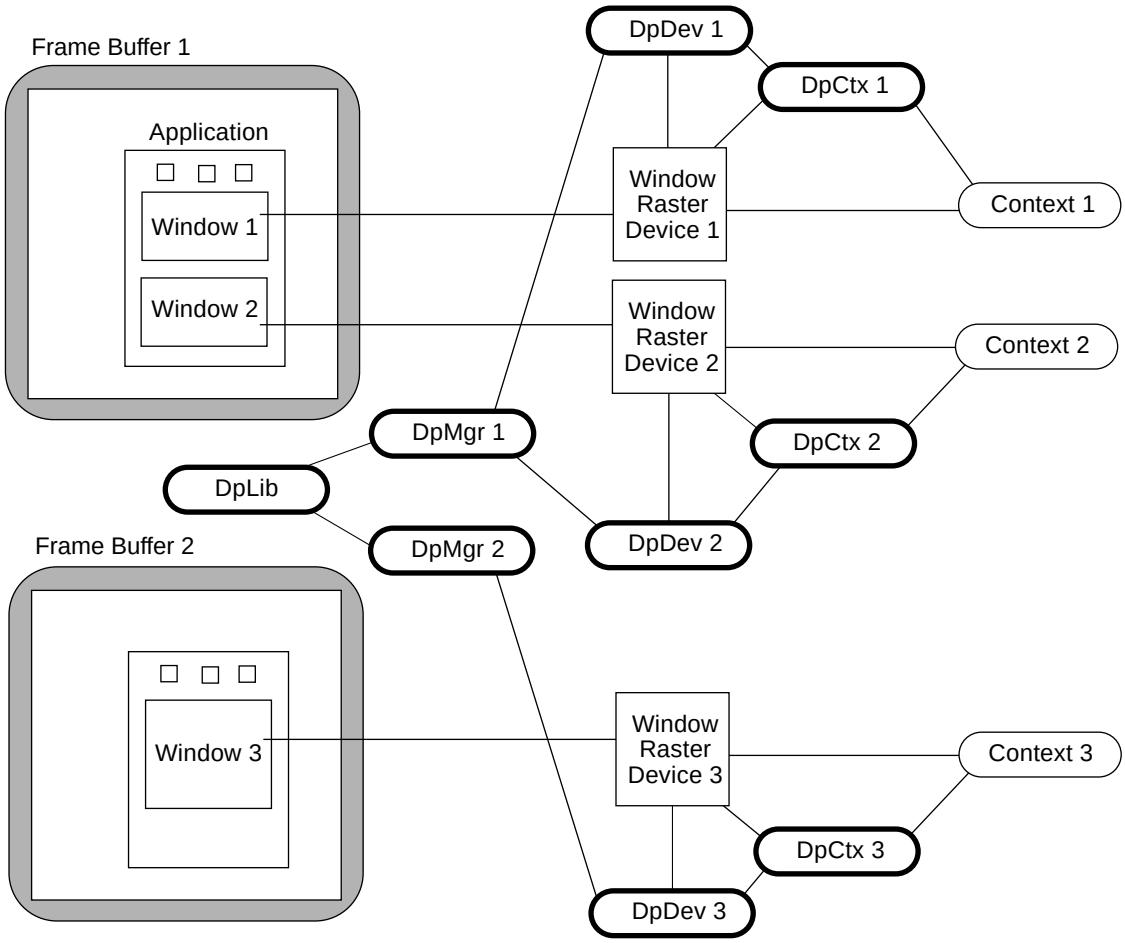


Figure 2-8 Device Pipeline Architecture: DpLib Object

2

Software Pipeline and Pipeline Switching

For all geometry primitives, XGL includes a complete software implementation of the top two layers of the rendering pipeline interfaces. Device pipeline implementors writing for a specific graphics hardware device can choose to interpose their own functions for the interfaces that exist at the top two pipeline layers, or they can let the XGL-supplied software pipeline perform the tasks. Even if the device pipeline provides its own implementation of pipeline interfaces, it can call the software pipeline at the LI-1 or LI-2 layers to continue processing the data.

The lowest layer of loadable interfaces, which is responsible for writing pixels to the device, is device dependent. This layer does not exist in software, although there is a set of utilities in software that implements most of LI-3. A device pipeline must include functions for this lowest layer of the pipeline.

Handling Context State Changes

An application program can cause the state of a single XGL Context object to change in two ways:

- The application can set attributes to explicitly change the state of an XGL Context object.
- The application can change objects associated with the Context, such as a Transform object. This indirectly causes Context state to change.

Changes in the state of a Context object resulting from attribute setting on the Context or on an object associated with the Context are passed directly to the device pipeline by the XGL device-independent code. However, because of the various models of hardware context switching, changes to Context state resulting from intraprocess Context switching (in other words, the application switches the XGL Context it is using to render) are left to the device pipeline implementor to manage.

First Things First: What Is Context?

Before examining how the XGL core handles Context state changes, it might be helpful to look at the different uses of the term *context*. From the point of view of XGL, context can refer to the XGL API Context object, to the context of a UNIX® process, or to a hardware context. In a broad sense, *context* refers to a set of persistent state that controls an executing entity. We might apply this definition to the types of context as follows:

XGL Context	The set of state that controls rendering of XGL primitives, such as line color or transforms.
Process context	The set of state that controls a UNIX process, such as the program counter, the signal mask, or file descriptors. This state also includes memory mapping for devices.
Hardware context	The set of state that controls the rendering on graphics accelerators, for example line color or raster operation register values.

The problem for the device pipeline writer is to map these different sets of state to the graphics hardware. This mapping is potentially complex, since the following points should be considered:

- The graphics hardware may have one or more sets of state, which provide one or more hardware contexts.
- There are one or more UNIX processes using the graphics hardware via XGL. There is one special process, the X window server, which is also using the hardware.

  DGA and the segment driver help maintain consistency between each of the UNIX processes and the hardware contexts. Note that the kernel may context-switch a process at any time.
- Each of the UNIX processes may be an XGL program that opens one or more application windows via the X server. Each window is bound to an XGL Device, and each Device may have many XGL Contexts.

Thus, the challenge for the device pipeline writer is to map an arbitrary number of XGL Context objects onto the number of hardware contexts supported by the graphics accelerator.

Explicit XGL State Changes

Operations required for keeping track of the XGL Context attribute state are handled by the Context object in a device-independent manner. These operations include the getting and setting of attribute values and the storing of values. Information on attribute changes is passed directly to the pipelines. When the device pipeline is first associated with an XGL Context object, it gets the current values of the attributes and sets up the hardware. During program execution, the pipeline is notified of attribute changes as the changes occur, and it has the option of updating the hardware each time changes occur or of noting that changes occurred and updating the hardware at a later time.

The mechanism that notifies the pipeline of attribute changes is an array defined in the device-independent XglDpCtx object and managed by the pipeline in its device's specific XglDpCtx object. This array serves as the means of communication between the device-independent code and the device pipeline.

Intraprocess State Changes

When an application switches from drawing through one Context to drawing through another, state information must be saved in the graphics pipeline and the hardware. Because the XGL system cannot know what the state requirements for a particular hardware device are, how many hardware contexts are available, or how expensive it is to swap hardware contexts, XGL has left it up to the device pipeline implementor to handle state changes resulting from intraprocess context switching.

XGL defines the basic components of the device pipeline, but the pipeline writer must complete the task of mapping XGL state to the hardware. For example, a pipeline for a frame buffer with one hardware context might decide that it needs a concept of a current XGL Context. To implement this, it may use one of the device pipeline interface objects to keep track of which Context object is the current Context.

The pipeline architecture provides four places for the device pipeline implementor to insert device-dependent information for a specific hardware device. Depending on the requirements of the hardware, the device pipeline writer can include data in the pipeline interface objects:

- DpCtx object - Include data relevant to state information
- DpDev object - Include data relevant to the device or window
- DpMgr object - Include data relevant to the physical device
- DpLib object - Include data relevant to the pipeline library as a whole

By providing a variety of locations for device-dependent information, the XGL architecture has given the device pipeline writer the flexibility to support different hardware state models.

Window System Interaction

The XGL system uses Sun's DGA technology to accelerate XGL applications that are running under the OpenWindows server. DGA is a set of mechanisms that enables OpenWindows client processes to directly drive a graphics device when the client and server are on the same machine.

The current implementation of DGA uses shared memory as the means of coordinating information between the client and the server. The shared memory contains information about the window, such as the window's size and position on the screen. DGA includes a client library of functions that enable the client program to get information about the window from the shared memory and to lock the window so that the client can render. The client library is part of the OpenWindows product and includes functions that provide access to the window, manage rapid color map changes, perform multibuffering, support windows with backing store, and handle software cursors when rendering.

To provide the device pipelines with a way to manage window system interaction and with an interface to the DGA client library, the XGL architecture provides the XglDrawable object. The XglDrawable conceptualizes the sharing of a device with another entity, most often the window system, but possibly also a Memory Raster or a stream device. If the other entity is the window system, the XglDrawable manages the sharing of information through DGA (for local rendering) or through PEXlib / Xlib (for remote rendering). For example, when the window clip list changes, the two entities that are sharing the device must each be made aware of the changes and must cooperate to manage the changes.

2

The XglDrawable object provides a high-level abstraction of the window for the variety of devices that XGL has to deal with. The goals of the XglDrawable are to:

- Encapsulate the DGA interface within the XglDrawable object. This allows the specifics of the sharing of the device and the other entity to be hidden from the pipeline.
- Hide window system dependencies and actual device interaction from the XGL core.
- Hide the differences in client- and server-mode DGA interactions from the device pipelines.
- Hide the handling of backing store from the device pipelines.

The XglDrawable class derives to objects that reflect the type of device the application is rendering on. For a Window Raster Device, the XglDrawable object embeds the DGA client library, `libdga`, as an underlying layer, thus providing the device pipelines with the DGA window-locking mechanism for rendering and enabling the pipelines to determine whether the window clip list has changed. The Memory Raster Drawable object contains information on the user clip list and on the depth of the window but does not embed DGA calls or workstation clip list information.

XglDrawable objects can be created for Xlib rendering, for rendering to a backing store, and potentially for other types of rendering, such as rendering to a raw frame buffer. The instantiation of the appropriate XglDrawable object for a device is handled automatically by the XGL device-independent code. Once the object is instantiated, the XGL core does not need to check the device or Drawable type before invoking a window system operation.

Color

XGL supports both indexed and RGB color models, and it allows an application to use either color model without requiring the application to take into account the color model of the hardware that it is running on. XGL manages the mapping of color values to the underlying hardware for the application. At device creation time, XGL determines the real color type of the device and maps the color type that the application specified for the API Device object to the actual color type of the hardware. In this mapping, XGL must take into account these four cases:

- Indexed application color model on indexed hardware device
- Indexed application color model on RGB hardware device
- RGB application color model on indexed hardware device
- RGB application color model on RGB hardware device

The mapping consists of translating XGL colors into pixel values (in the case of the X window system, into X pixel values), such that the final device color format represents a color as close as possible to the original XGL color. The process of color translation into X pixel values is shown in Figure 2-9.

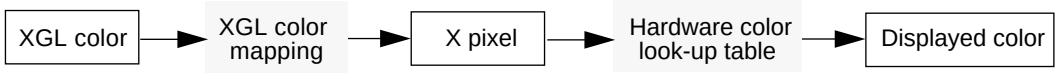


Figure 2-9 XGL Color Translation

For window rasters, the translation scheme depends on the hardware but the goal is the same: the visible color should be as close as possible to the requested color.

To manage the relationship between the XGL Color Map object (defining the application color type) and the XGL window raster, XGL provides the XglCmapDrawable object. This object encapsulates color handling and hides the sharing of colors between XGL and the window system.

The XglCmapDrawable object is instantiated by the XGL core during Window Raster creation. It is associated with both the Window Raster object and the Color Map object. Note that, unlike the XglDrawable object, there is no need for XglCmapDrawable objects for Memory Raster devices or Stream devices.

2

XGL Class Structure 3

This chapter discusses the XGL class hierarchies. In this chapter, device-independent classes are referred to as DI classes, and pipeline classes are referred to as *device pipeline* (Dp) classes and *software pipeline* (Swp) classes.

Overview of the XGL Class Structure

The XGL class hierarchy can be thought of as providing three sets of classes:

- Classes for implementing API objects
- Internal classes used for device-independent utilities
- Classes subclassed by device pipeline implementors for device pipelines

The root of the XGL class tree is XglDbgObject. All device-independent, device pipeline, and software pipeline classes derive from this class. This class also assigns each instantiated object an object ID for debugging purposes.

XglDbgObject has a number of immediate subclasses. A top-level view of the XGL class structure is shown in Figure 3-1 on page 42. In this diagram, pipeline classes are shown with dark borders, the base class of the API hierarchy is shown as a filled oval, and the internal utility classes have thin borders.

3

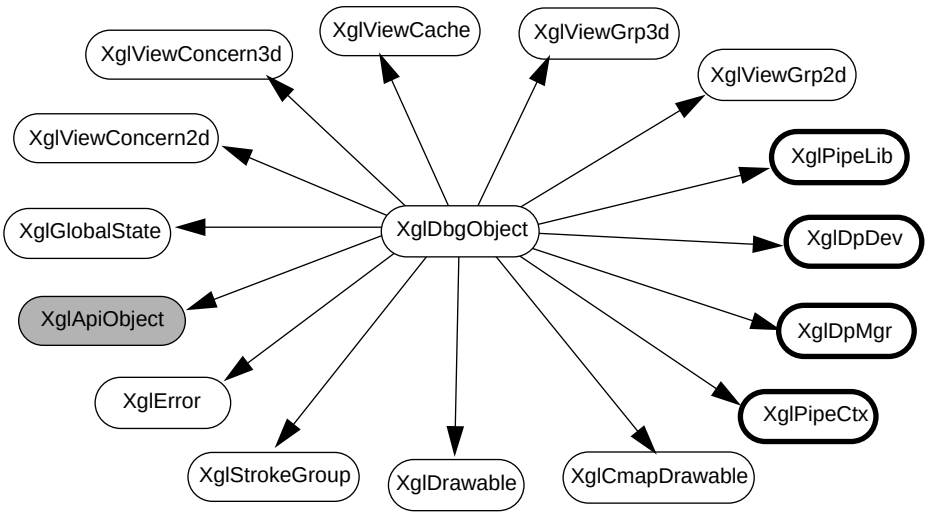


Figure 3-1 Top-Level View of the XGL Class Hierarchies

The XGL device-independent classes and the device pipeline and software pipeline classes are described in more detail in the sections that follow and in subsequent chapters. See Chapter 6, “Error Handling” for information on XGL error handling.

Device-Independent Classes

The device-independent classes handle operations that are done in a device-independent manner on behalf of the device pipelines and software pipeline in response to requests from the API. Device-independent classes are defined by the XGL device-independent code. There are two general categories of DI classes: the API classes and the internal utility classes.

Classes That Implement the XGL API

The XGL API presents the C programmer with objects that are abstractions of graphics resources. Internally, XGL implements these objects using a hierarchy of C++ classes. Some classes in this hierarchy become API objects used by the application; other classes are not made into objects but are simply subclassed to provide classes that become API objects. The complete API class hierarchy is shown in Figure 3-2.

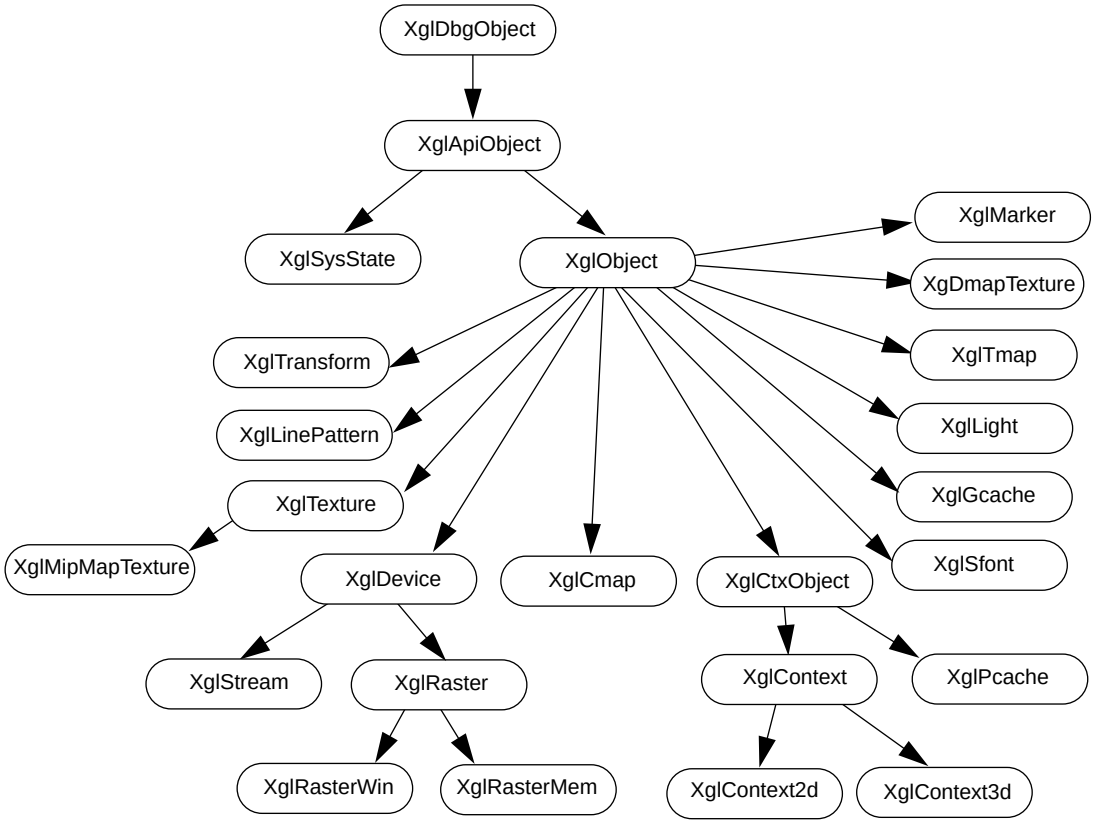


Figure 3-2 API Class Hierarchy

The creation of the System State object is managed by the XglGlobalState object. All other API objects are instantiated through the System State object. The System State object maintains a list of all API objects created in the XGL system and is responsible for destroying API objects during `xgl_close()`. See Chapter 4, “Object Interactions” for information on object instantiation through the System State object. For information on the API objects, see the *XGL Programmer’s Guide* or the *XGL Reference Manual.*

The classes in the API hierarchy that do not become API objects are:

XglApiObject
Contains the object type and pointer to the application data. The setting and getting of application data (API attribute `XGL_OBJ_APPLICATION_DATA`) is handled in this class.

XglObject
The base class of the API classes with the exception of the System State class. XglObject manages the user list for the API objects. The interface for user list message receiving is defined in this class and is overridden by the API object classes that do processing based on received messages. XglObject also handles object destruction.

XglCtxObject
Defines the Context hierarchy and the Pcache object.

XglDevice
Contains device-independent data and operations that are common to all devices. This object constitutes the device-independent part of a device object and contains a pointer to the device-dependent part of the device object. For information on the components of the Device object, see Chapter 4, “Object Interactions”.

XglRaster
Contains the raster-specific definition of a device. XglRaster also contains a pointer to the device-dependent part of the Raster object. This class is the parent class of the XglRasterWin and XglRasterMem classes.

XglContext
Contains the device-independent part of the Context object. XglContext contains device-independent data, some cached data that all device pipelines and the software pipeline share, and functions to set and get

Context attributes. XglContext is the parent class for the XglContext2d and XglContext3d classes. For information on the components of the Context object, see Chapter 4, “Object Interactions”.

Classes That Provide Internal Utility Functions

The device-independent utility classes provide a variety of functions. For example, several of these classes work together to provide a mechanism that enables the passing of view model information from the application to the device pipeline. Briefly, the DI utility classes provide the following functions:

XglGlobalState
The XglGlobalState class handles device pipeline and software pipeline loading and maintains the handles to the shared objects. XglGlobalState also handles some of the initial interactions between the device-independent classes and the device-pipeline classes. The XglGlobalState class instantiates the XglGlobalState object during `xgl_open()` just before the System State object is created.

XglViewCache
The XglViewCache class is the base class of the view model derived data facility. It is the abstract class from which XglViewCache2d and XglViewCache3d are derived. XglViewCache defines the data and functions common to the 2D and 3D classes. XglViewCache2d and XglViewCache3d consist of cached view model derived data and functions for evaluating the data when the pipeline requests it. Figure 3-3 shows the class hierarchy for the view cache class. For more information on the view modal derived data facility, see Chapter 5, “Rendering and Handling State Changes.

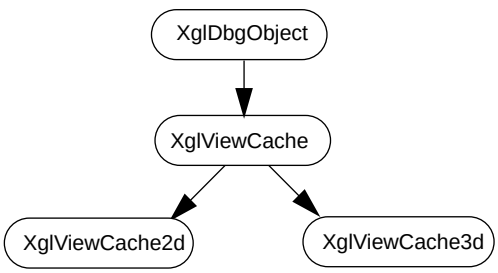


Figure 3-3 View Cache Class Hierarchy

XglViewGrp{2,3}d

XglViewGrp{2,3}d is the base class that defines pointers to evaluation functions in the cache of view model derived data. It is the abstract class from which XglViewGrp{2,3}dConfig and XglViewGrp{2,3}dItf are derived. Figure 3-4 shows the class hierarchy for the view group class in the 2D case.

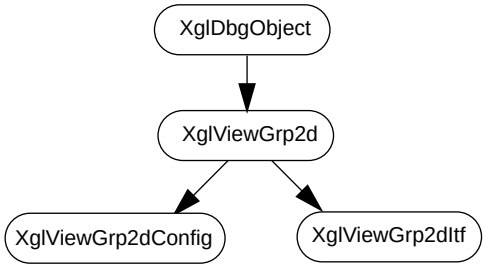


Figure 3-4 View Group Class Hierarchy

XglViewGrp{2,3}dItf provides the interface to the view model derived data from device and software pipelines. An object of this class consists of functions that describe when derived data have changed and functions for getting evaluated individual derived data items.

XglViewGrp{2,3}dConfig describes the configuration of the view model derived data for geometry entering LI-1 from a particular coordinate system. An object of this class consists of flags in a table for individual derived data items adjusted for a coordinate system that can be other than model coordinates.

XglViewConcern{2,3}d

XglViewConcern{2,3}d registers the collection of concerns for each coordinate system from which geometry can enter LI-1. Upon entering LI-1, a primitive for a pipeline cares about only a limited selection of derived data items. Objects of this class store this selection for each coordinate system.

XglDrawable

The XglDrawable class provides a public interface for window system interaction. This class maintains information about the window size and clip list, as well as information about the device. XglDrawable subclasses to drawables for Window Raster objects, Memory Raster objects, Xlib / PEXlib connections, and backing store objects.

The base XglDrawable class serves as a repository for shared internal data, such as window dimensions and clip list information, as well as providing default or common operations for the derived classes. It also provides the public interface for the hierarchy. Actual window system interactions are implemented in the derived classes and are transparent to the device pipelines.

Creation of the appropriate Drawable class object is done by the XGL core through the XglDrawable class during XGL device creation, based on the descriptor that is passed in through the API or attributes that are set internally (for backing store mode).

XglCmapDrawable

The XglCmapDrawable class encapsulates color sharing between XGL and the X window server.

XglStrokeGroup

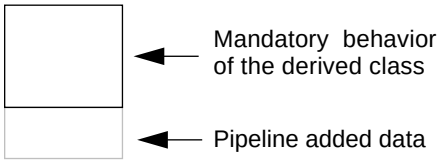
The XglStrokeGroup class maintains the attributes used for multiplexing line drawing to handle different primitives. The Context object maintains one XglStrokeGroup object for lines, markers, text, edges, and hollow polygons. For more information on stroke groups, see Chapter 5, “Rendering and Handling State Changes”.

3

Device Pipeline Classes

The pipeline hierarchies provide predefined interfaces between the device-independent code and the device pipelines. These interfaces allow the XGL device-independent code to interact with the device pipeline code in expected ways. For each specific device pipeline implementation, the device pipeline writer must subclass a device-dependent class from each of the four pipeline class hierarchies. The objects instantiated from the device-specific subclasses will then provide the functionality that the XGL device-independent code expects.

All member functions for device-dependent internal object management are defined in the base classes provided by the XGL device-independent code. When subclassing a device pipeline class, the device pipeline writer can add member data and functions as needed.



The four device-dependent pipeline class hierarchies are listed in Table 3-1.

Table 3-1 Device Pipeline Class Hierarchies

Name	Definition	Description
XglPipeLib	Hierarchy for the pipeline library objects	This hierarchy represents the loaded shared library of a device or software pipeline. It handles the creation and destruction of the XglDpMgr subclass of objects.
XglDpMgr	Hierarchy for the device pipeline manager	This hierarchy represents a category of devices, such as frame buffers. In the case of multiple frame buffers, a XglPipeLib object can choose to maintain multiple XglDpMgr objects, each of which maps to a physical frame buffer. This hierarchy handles the creation of the XglDpDev subclass of objects.

Table 3-1 Device Pipeline Class Hierarchies

Name	Definition	Description
XglDpDev	Hierarchy for the device-dependent part of the Device object	This hierarchy holds the device-dependent elements of an XglDevice object. It also manages the creation and manipulation of the device pipeline-context objects.This hierarchy is device-specific and is not subclassed by the software pipeline.
XglPipeCtx	Hierarchy for the pipeline-context objects	This hierarchy represents the interfaces for the loadable interface layers of the device pipelines and the software pipeline.

The following sections provide information on the pipeline hierarchies.

Pipeline Library Class Hierarchy

A pipeline library object maps to to a unique `.so` shared library. Thus, for each shared library (for example, `libxglcfb.so`) that is loaded into the XGL environment, there is an XglPipeLib subclassed object that represents it.

The base class of the pipeline library hierarchy is XglPipeLib. XglPipeLib subclasses to the device pipeline library (XglDpLib) class and the software pipeline library (XglSwpLib) class. XglPipeLib simply serves as a general category for these two specific shared library classes and has no member functions of its own. Individual pipeline implementations derive to device pipeline-specific objects, such as XglDpLibGx. Figure 3-5 shows the pipeline library hierarchy.

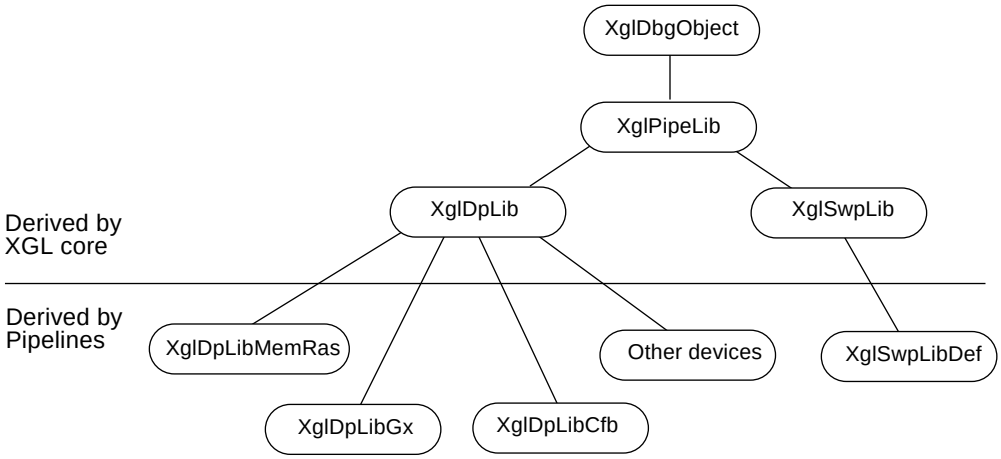


Figure 3-5 Pipeline Library Class Hierarchy

The primary characteristics of the device pipeline library and software pipeline library base classes are as follows:

XglSwpLib

The XglSwpLib object represents the unique software pipeline shared library and is the base class for the software pipeline class.

When the software pipeline shared library is loaded, the XglSwpLib object is created by a call to `xgl_create_PipeLib()`. The handle to the XglSwpLib object is maintained by the XglGlobalState object. There is one XglSwpLib object for the software pipeline in the XGL system, and it is assumed that there is only one software pipeline. The software pipeline is the default pipeline and is provided with the XGL product.

The XglSwpLib class is responsible for creating the software pipeline-context objects during Context creation. The XglSwpLib class can be used to store state information that can be shared by all software-context objects.

XglDpLib

The XglDpLib object is the base class for the XglDpLib classes in the device pipelines. This class maps to a loaded device pipeline, and there is one loaded device pipeline per device type. XglDpLib is also responsible for the creation of the device pipeline manager object. In the case of frame buffers, XglDpLib allows more than one XglDpMgr (each of which represents a physical frame buffer) to share hardware or software resources. When the device pipeline shared library is loaded through the XglGlobalState object, an instance of the derived XglDpLib object is created by a call to `xgl_create_PipeLib()`.

Note that since the XGL core does not control the actual number of XglDpMgr objects created, it is the device pipeline's responsibility to destroy any existing XglDpMgr objects during the destruction of the XglDpLib object when this is invoked by the XGL core.

Device Pipeline Manager Class Hierarchy

The XglDpMgr class is responsible for the creation and management of the device-dependent part (XglDpDev) of a Device object. XglDpMgr allows multiple XglDpDev objects to share the physical resources of a device. XglDpMgr is the base class for the XglDpMgr subclasses defined in a pipeline implementation.

In the case of a window raster device, each instance of an XglDpMgr subclass represents a physical device (frame buffer), and there is one XglDpMgr object per physical device. The frame buffers can be the same type (for example, both GX) or different types. If there are multiple devices of the same type on a system, there will usually be multiple instantiations of the same XglDpMgr subclass, all of which map to the same device pipeline. However, since the creation and destruction of all XglDpMgr objects is handled internally by the device-dependent XglDpLib object, a device can choose to have only one XglDpMgr object for more than one frame buffer of its type.

If the devices are of different types, the XglDpMgr object corresponding to each device type will be created by the XglDpLib unique to the shared library. Figure 3-6 shows the device pipeline manager class hierarchy.

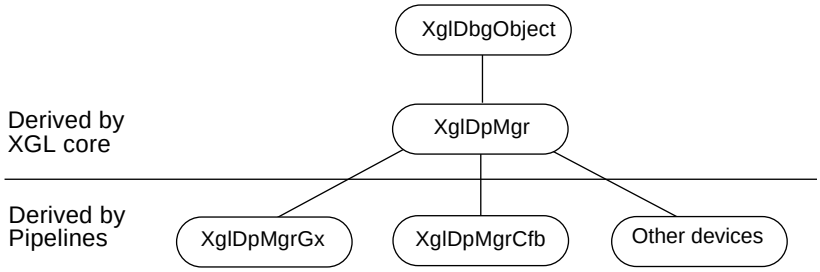


Figure 3-6 Device Pipeline Manager Class Hierarchy

Device-Dependent Device Class Hierarchy

XglDpDev is the base class for the device-dependent part of the Device object. A pointer to the XglDpDev object is stored by the device-independent Device object. XglDpDev has the following functions:

- It creates the device pipeline-context objects.
- It performs device-dependent device operations, such as propagating changes of the device-independent Device object to the device pipeline.

XglDpDev is created as part of the Device object creation process and remains attached to the Device object for the lifetime of the Device instance. XglDpDev was designed to isolate the device-dependent operations from the device-independent operations. Each pipeline implementation must define the actual device-specific operations for the device.

If an application has multiple windows using the same underlying frame buffer, the XglDpMgr object representing that frame buffer will create multiple XglDpDev objects. If an application runs on a system with more than one frame buffer and creates multiple windows on each frame buffer, the XglDpMgr object representing each frame buffer will create the XglDpDev objects for the application windows. For an illustration of the device pipeline objects created for a multi-headed system, see Chapter 2, “Overview of the XGL Architecture”.

In Figure 3-7, XglDpDevWinRas and XglDpDevMemRas contain the abstract class interfaces provided by the XGL core. The pipeline implementation derives to XglDpDevCfb, XglDpDevGx, etc., to define the actual device-dependent operations.

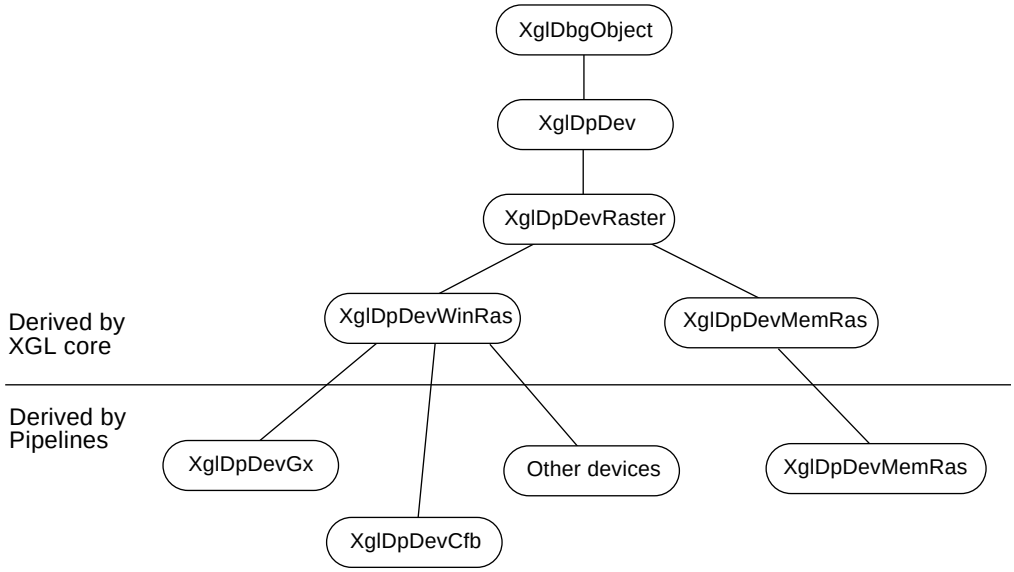


Figure 3-7 Device-Dependent Device Class Hierarchy

Pipeline-Context Class Hierarchy

The base class of the pipeline-context hierarchy is XglPipeCtx. The derived classes in this hierarchy, XglPipeCtx2d and XglPipeCtx3d, represent the pipeline interfaces for Context operations. The software and device pipelines subclass from these two classes to implement the actual 2D and 3D primitive operations.

Each Context object links with two XglPipeCtx subclassed objects: one for the software pipeline (for example, XglSwpCtx3d) and one for the device pipeline (for example, XglDpCtx3dCfb). Figure 3-8 on page 54 illustrates the pipeline-context hierarchy.

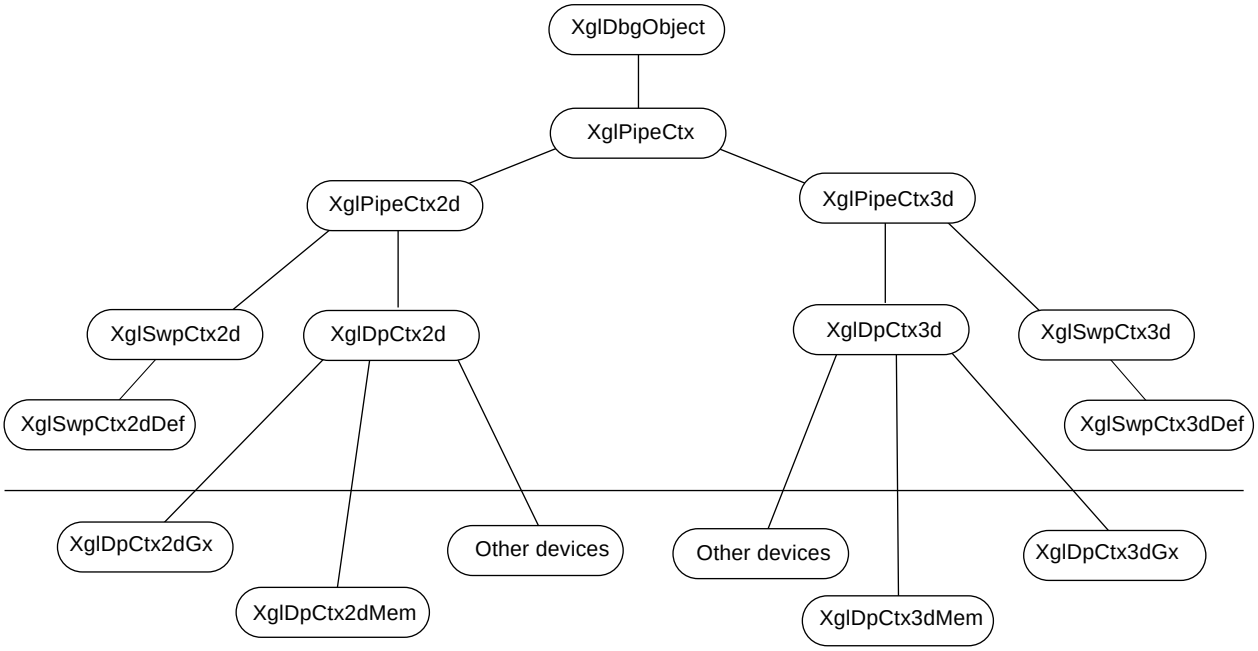


Figure 3-8 Pipeline-Context Class Hierarchy

Pipeline-Context Objects

The software pipeline-context object is created by XglSwpLib during XglContext creation time and remains attached to the Context object for the lifetime of the Context object.

The device pipeline-context object is created and maintained by a subclass of XglDpDev during the first association of a Context and a Device. The device pipeline-context object remains associated with the Context object until the Context switches devices. When a Context object is destroyed, the System State object destroys the associated pipeline-context objects for all existing devices.

The interfaces for the XglPipeCtx hierarchy are defined in an array of function pointers to device pipeline or software pipeline renderers. The function pointers represent the primitives for Context operations at the LI-1, LI-2, and LI-3 layers.

Classes for Internal Data Storage

The internal data types are represented by the following C++ classes:

XglPrimData
The XglPrimData class formats API geometric data to the XGL internal point type in the software pipeline. It includes an array of several *levels* of point list data (contained in the XglLevel subclass) that can be created when data is moved down through the graphics pipeline (in other words, transformed points are stored at a different level than the original points). Briefly, this class includes member functions that:

- Allocate and free memory used for containing geometric data, and copy and convert geometric data into the form appropriate for a particular stage of the geometric pipeline.
- Set and test the per-facet or per-primitive flags for hollow flags, global edge flags, vertex edge flags, pick information, and silhouette edge flags.

XglConicData2d / XglConicData3d
The XglConicData classes format conic data (in circles, arcs, ellipses, or elliptical arcs) to the appropriate internal point type. Like the XglPrimData class, these classes include an array of levels of point list data (contained in the XglConicList2d and XglConicList3d subclasses) that can be created when data is moved down through the graphics pipeline.

XglRectData2d / XglRectData3d
The XglRectData classes format rectangle data to the appropriate internal point type. Like the XglPrimData class, this class includes an array of levels of point list data (contained in the XglRectList2d and XglRectList3d subclasses) that can be created when data is moved down through the graphics pipeline.

XglPixRect
The XglPixRect class manages pixel data. A PixRect is an abstraction of a 2D rectangular array of pixels. The XglPixRect class subclasses to classes that represent memory-based PixRect objects.

For more information on the internal point type classes, see the *XGL Device Pipeline Porting Guide*.

3

Object Interactions 4

This chapter provides information on XGL objects. It includes information on the following topics:

- API object instantiation
- Pipeline loading and pipeline object instantiation
- Communication between objects
- Object destruction

Opening XGL

The API operator `xgl_open()`initializes XGL. This operator first checks that the XglGlobalState object does not already exist (to prevent multiple `xgl_open()` calls). If this is the case, the `xgl_open()` routine instantiates the Global State object. The Global State object maintains the state of the XGL environment and is responsible for the loading and manipulation of the pipeline shared library objects and the corresponding XglPipeLib objects. There is only one Global State object in the XGL environment.

The XglGlobalState object loads the software pipeline shared library and then calls the function `xgl_create_PipeLib()` (defined in the software pipeline shared library) to create the XglSwpLib object. The XglSwpLib object defines the member functions necessary to create and destroy the software pipeline subclasses of the DI provided classes. For more information on pipeline loading, see “How the Pipelines Are Created and Managed” on page 64.

The Global State object also creates the System State object. The System State object handles API object creation and destruction, error mode setting, and internal error reporting. In future releases, it may be possible to have more than one System State object. If multiple System States are allowed, the Global State object will manage them.

Note – The software pipeline is loaded when XGL is opened so that the open fails if there is no software pipeline.

How API Calls Are Mapped to XGL Internal Calls

From the API perspective, all API-visible XGL objects are device independent. The application manipulates XGL objects and executes primitives via C object handles and C function calls. The C function calls and object handles are mapped to the C++ internal code by a set of C wrappers that provide the translation between the C API and the C++ device-independent classes and objects. For example, the application might ask to have a 2D Context created:

```
ctx = xgl_object_create(SysState, XGL_2D_CTX, NULL, NULL);
```

This call is mapped in `XglWrapApi.cc` to the C++ internal call:

```
obj = (XglObject*)((XglSysState*) sys_state)->createContext2d();
```

Each API object has a wrapper function that maps its object-specific operators and attribute set and get calls to internal calls. For example, an API request to set the line style in a 2D Context object:

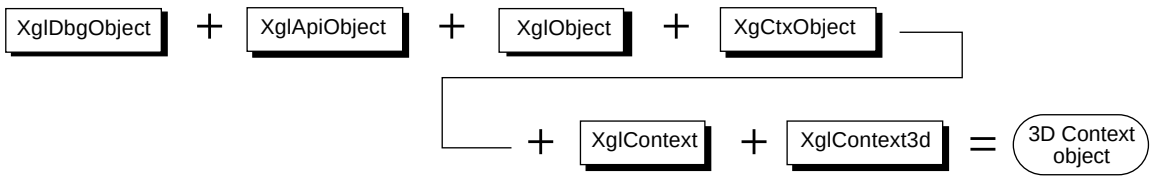
```
xgl_object_set(ctx, XGL_CTX_LINE_STYLE, XGL_LINE_PATTERNED, NULL);
```

is mapped to the internal call:

```
ctx2d->setLineStyle(<argument>);
```

Instantiation of API Objects

As mentioned in Chapter 1, an object consists of member data and the methods (or functions) needed to operate on that data. A class is a container for data and methods common to a set of similar objects. An instance of a class inherits the data and methods of each of its parent classes, starting from the top of the class tree and adding the data and methods from each of the derived classes. Thus, for example, a 3D Context object has the following inheritance:



The 3D Context object inherits the following data and methods from its parent classes:

XglDbgObject	This object is intended for debugging purposes only.
XglApiObject	Adds fields for the object type and application data.
XglObject	Adds mechanisms to enable information on state changes to be communicated between objects.
XglCtxObject	Adds data and methods to enable the use of the Pcache object.
XglContext	Adds data and methods shared between 2D and 3D Context objects. For example, surface front face attributes are available to both 2D and 3D objects.
XglContext3d	Adds 3D Context object-specific data and methods, such as attributes for depth cue color.

System State Object and API Object Lists

All API objects are instantiated through the System State object. The object creation call is routed by the wrapper function to the System State object, which instantiates the object. The System State object keeps track of all API objects in two lists: a list of pointers to Device objects and a list of pointers to all other API objects. When the System State object instantiates an API object, it adds a pointer to the object to the list.

For most API objects, object instantiation is performed in a reasonably straightforward way, with the System State object invoking object instantiation and the object constructor initializing default values. However, object instantiation for the Device object and the Context object is more complex, since each of these API objects is actually composed of more than one internal object. The sections that follow discuss the components of the Device and Context objects.

4

What Is a Device Object?

From the perspective of the XGL API, a Device object is an abstract entity that represents a drawing surface. From the internal perspective, however, a Device object is actually composed of two objects that are linked together throughout the lifetime of the objects. The components of the Device object are:

- A device-independent object subclassed and instantiated from one of the leaves of the Device class, such as XglRasterWin.
- A device-dependent object subclassed and instantiated from a corresponding leaf of the XglDpDev class, which in the case of a window raster device might be XglDpDevGx. The XglDpDev subclassed object is created as part of the Device object creation process and remains associated with the Device object for the lifetime of the Device instance.

The two ways of looking at a Device object are illustrated in Figure 4-1, using a window raster device as an example.

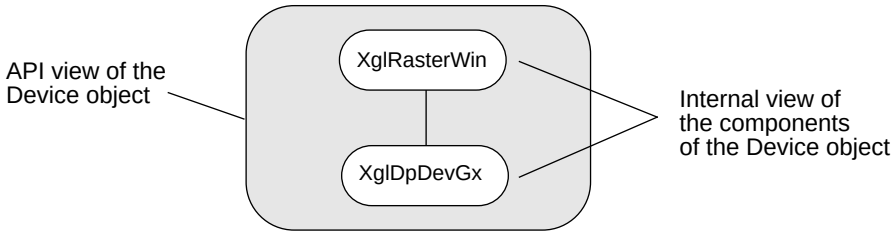


Figure 4-1 Components of the Device Object

The Device object was designed with separate device-independent and device-dependent components to isolate the Device object's device-dependent operations from its device-independent operations. In this way, each pipeline implementation can define the actual device-specific operations for the device.

Instantiation of a Device object is a complex process involving numerous steps. See "How the Pipelines Are Created and Managed" on page 64 for information on this process.

What Is a Context Object?

The Context object is responsible for rendering and storing attribute values. As with the Device object, the conceptual API view of the Context object is different from its actual internal representation. From the perspective of the application, the Context object is a single entity that represents the graphics pipeline. Internally, however, the Context object has several component objects. When the API Context object is initially created, it is composed of the following two objects:

- A device-independent object subclassed and instantiated from one of the leaves of the Context class, such as XglContext3d.
- A device-dependent object subclassed and instantiated from one of the leaves of the XglPipeCtx class for the software pipeline, such as XglSwpCtx3d. The software pipeline-context object is linked with the Context object for the lifetime of the Context object.

Additionally, when the Device object is associated with the Context object, the following object becomes a component of the Context object:

- A device-dependent object for the device pipeline that is subclassed and instantiated from the leaves of the XglPipeCtx class that correspond to the already existing Device and Context, such as XglDpCtx3dGx.

Figure 4-2 shows a 2D Context object that has been associated with a GX frame buffer.

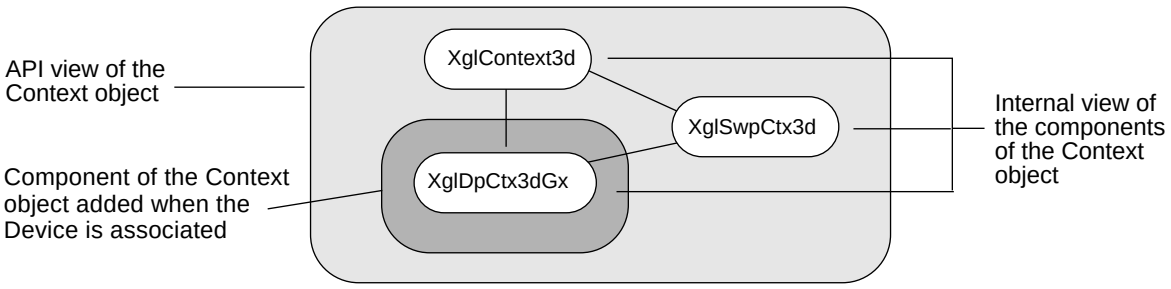


Figure 4-2 Components of the Context Object

Device and Context Association

The device pipeline-context object is created during the first association of a Context and a Device and corresponds to a specific Device-Context pair. When a Device and a Context are first associated, the device-dependent part of the Device object creates a device pipeline-context object (XglDpCtx) for the device. If a Device is set on a Context with which it was previously associated, the Context will disassociate from its current device and ask the Device to retrieve the handle to the pre-existing XglDpCtx object. If the Device object finds a pre-existing entry for an XglDpCtx object for this Context, it returns a pointer to the object to the Context.

Figure 4-3 illustrates the relationships between the various DI and Dp objects when the Device is first associated with the Context. It shows a Window Raster Device object, including its device-independent part (XglRasterWin) and its device-dependent part (XglDpDev), the device pipeline-context object, and the Device's list of existing XglDpCtx objects.

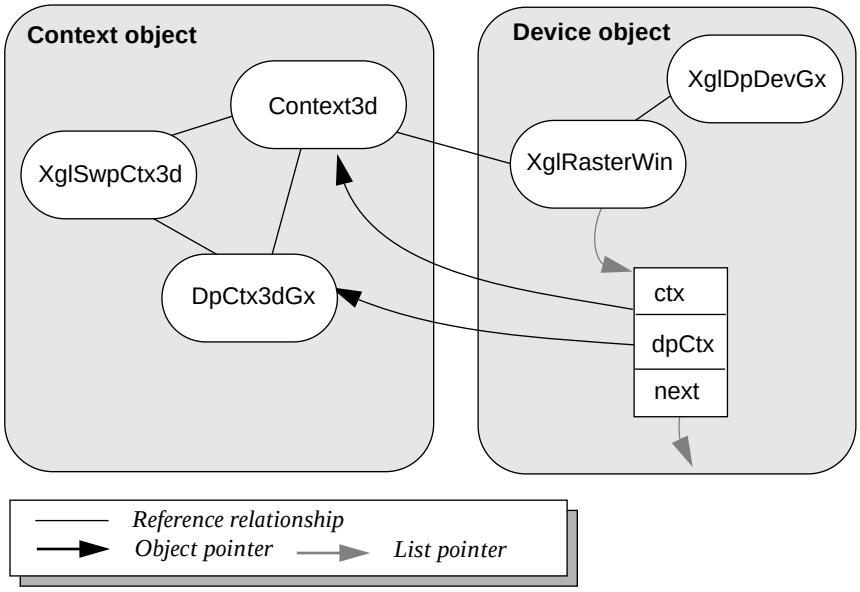


Figure 4-3 Device and Context Association

The Device object can be concurrently associated with more than one Context object. In this case, there is a device pipeline-context object created for each Device-Context pair. Figure 4-4 shows a Device object with two associated Context objects.

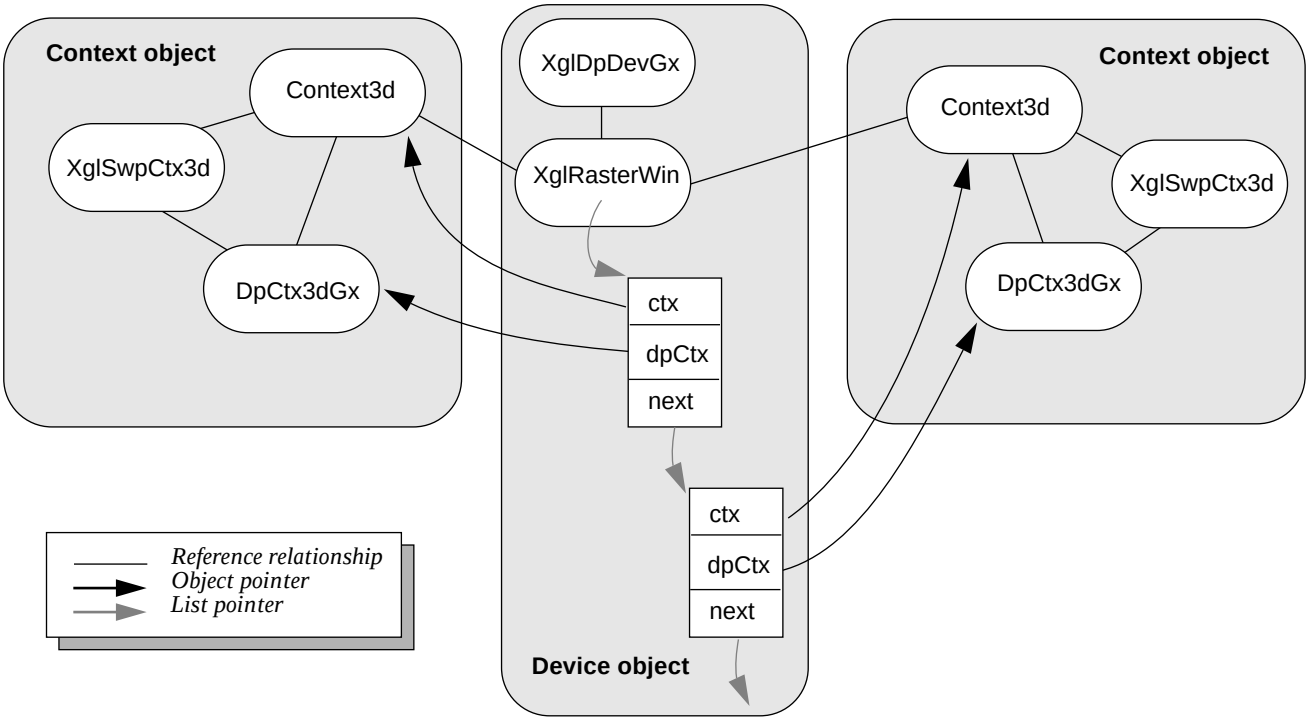


Figure 4-4 Device Association with Multiple Contexts

4

How the Pipelines Are Created and Managed

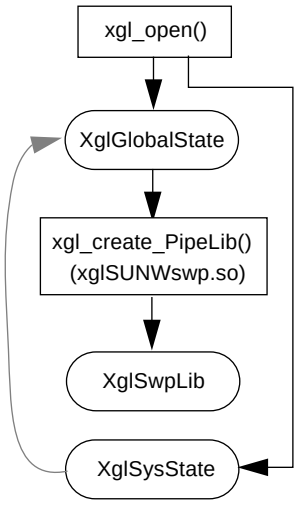
Pipeline creation is a dynamic process that occurs in phases as XGL is opened and initialized. The process can be summarized in the following general steps:

1. The XGL environment is set up.
2. A Device object is created.
3. A Context object is created.
4. The Device object is associated with the Context object.

The following sections describe this process. See Figure 4-5 on page 67 for an illustration of an instantiated runtime system.

Note – It is assumed that only one software pipeline exists in the XGL system, although the actual software pipeline can vary from one runtime environment to another.

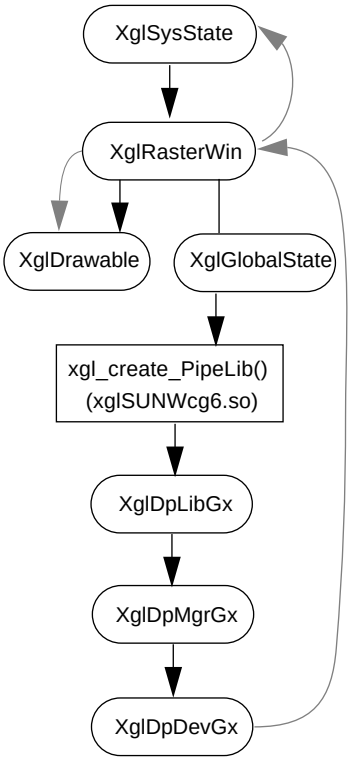
The XGL Environment Is Set Up



When an application calls `xgl_open()` to initialize XGL, the following set of events occurs. The events are illustrated in the diagram in the margin. Objects are represented by ovals; functions are in boxes. Black arrows represent control flow; shaded arrows represent pointers.

1. The `xgl_open()` call creates the XglGlobalState object, which is unique in the XGL system.
2. The XglGlobalState object loads the software pipeline (`xglSUNWswp.so`) through `dlopen()`.
3. The XglGlobalState object creates the unique XglSwpLib object for the software pipeline by calling `xgl_create_PipeLib()`, which is defined in `xglSUNWswp.so` and accessed through `dlsym()`.
4. The `xgl_open()` call creates the XglSysState object, and a handle to the System State object is returned to the Global State object. The System State object maintains two object lists: one for Device objects and one for non-Device API objects. During the creation of the System State, the predefined objects (line patterns, markers, stipple rasters) are created.

A Device Object Is Created



When the application calls `xgl_object_create()` to create a Device object (such as a window raster for a GX frame buffer), it passes in the device type and the window descriptor containing the X window identifying information. The illustration in the margin shows the creation of a Device object. The following events occur:

1. The XglSysState object initiates the creation of the device-independent part of the Device object. In the case of a window raster, for example, XglRasterWin is created.

2. When XglRasterWin is created, it calls XglDrawable::`grabDrawable()` to obtain an XglDrawable object. The XglDrawable `grabDrawable()` function determines the type of drawable required for the raster, and returns a drawable object of the appropriate type.

3. During the creation process of XglRasterWin (the device-independent Device object), the Device object asks the XglGlobalState object to create its device-dependent part.

4. To start the process of creating the device-dependent part of the Device object, the XglGlobalState object first traverses its XglDpLibList object list to determine if an object for the particular pipeline library already exists. If it finds a matching ID entry in the object list, the object exists, and the process proceeds to Step 6.

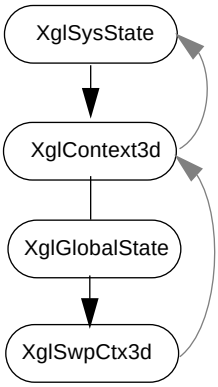
   If XglGlobalState does not find a match in its XglDpLibList, XglGlobalState loads the shared library using `dlopen()`. The name of the shared library is obtained from the Drawable object.

5. The XglGlobalState object creates the subclassed XglDpLib object for the pipeline being loaded, which is XglDpLibGx in this example. To do this, XglGlobalState calls `xgl_create_PipeLib()`, which is defined in the pipeline shared library and accessed through `dlsym()`. `xgl_create_PipeLib()` creates an instance of the pipeline derived XglDpLib class and returns a pointer. This pointer is then appended to the XglGlobalState's XglDpLibList object for future reference. Note that the XglDpLib object represents one pipeline.

6. The device pipeline derived XglDpLib object creates or retrieves an instance of the XglDpMgr pipeline derived object. There is one XglDpMgr object per device category, such as a frame buffer.

7. The XglGlobalState object then asks the XglDpMgr object to create the device-dependent subclassed XglDpDev object. An XglDpDev object is created for each new XGL Device. A pointer to this object is returned to the XglRasterWin object. A pointer to the XglRasterWin object is stored in the System State's list of existing Device objects.

A Context Object Is Created



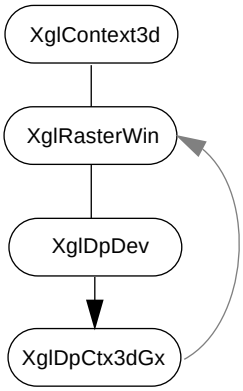
When the application calls `xgl_object_create()` to create a Context object, for example, XglContext3d, the following events occur:

1. The XglSysState object initiates the creation of the Context 3D object.
2. The Context object, through the Global State object, requests that the XglSwpLib object create a software pipeline-context object for the specific Context (in this example, XglSwpCtx3d). This object remains attached to the Context object for the lifetime of the Context object.

   The pointer to the initialized 3D Context object is stored in the System State object's list of existing non-Device API objects.

The illustration in the margin shows the creation of a Context object.

The Device Is Associated With the Context



When the application calls `xgl_object_set()` to associate the Device object and the Context object, the following events occur:

1. The Context object will disassociate from the current Device object, if any, and associate with the new Device object. (If the two Device objects are the same, no changes will occur, and the process is complete.)
2. The Context object will ask the Device object to retrieve the handle to the XglDpCtx object that corresponds to the specific Context. This object may have already been created by a previous association of the same Context and Device.
3. To retrieve the handle to the XglDpCtx object, the Device object will match against its list of XglDpCtx objects to determine whether an XglDpCtx subclassed object (e.g., XglDpCtx3dGx) for the specific Context and Device

pair already exists. If it exists, the handle to it is returned. Otherwise, the Device object will ask the subclassed XglDpDev object to create an XglDpCtx object for the Context and Device pair. The Device object will store a pointer to these objects for the lifetime of the Context object.

Figure 4-5 illustrates an instantiated runtime system. Black lines represent one-way or two-way relationships between objects. Shaded arrows represent pointers from an object list to the object.

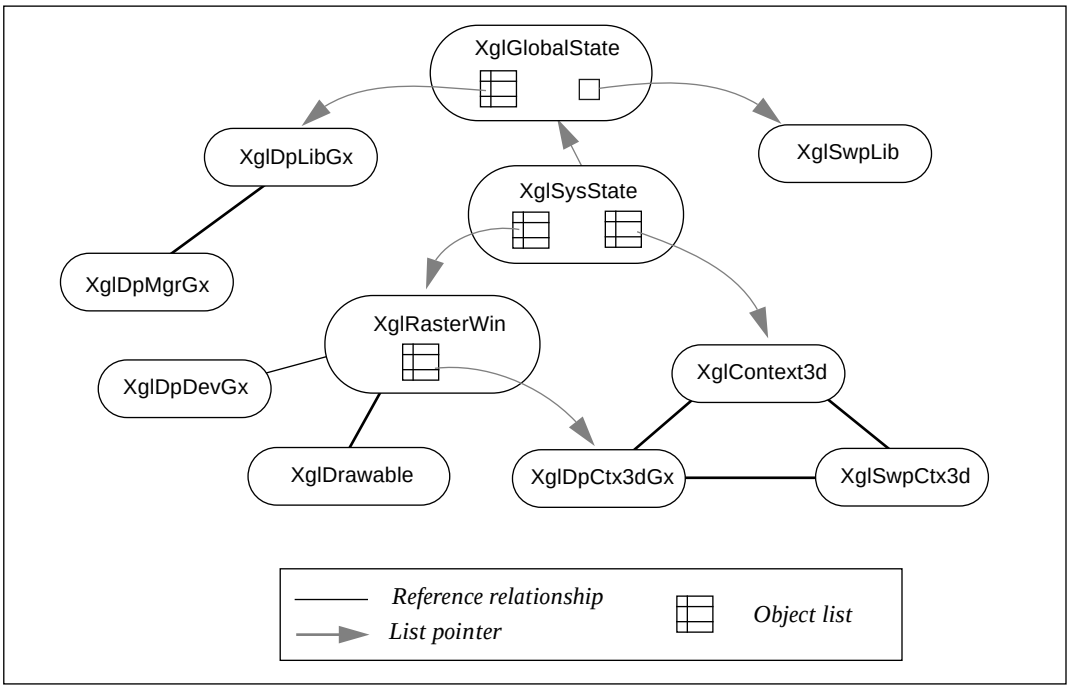


Figure 4-5 Pipeline Objects Instantiated at Runtime

4

Object Communication

This section describes how the internal relationship between XGL objects is set up and how messages are passed between objects.

API Object Relationships

During an XGL session, certain API objects are associated with other objects that serve as resources containing information relevant to rendering. This association between objects is established by the application's use of attributes whose values are pointers to objects.

Objects are associated with each other in the following manner: each object has a set of attributes, and one of these attributes may be a pointer to another object (or an array of pointers to objects). For example, during an XGL session, the application associates the Device object with the Context object, and it may also associate additional objects with the Context object, such as Memory Raster objects and Stroke Font objects. A relationship in which an object uses another object is referred to as a *using* relationship. Table 4-1 lists XGL objects that use other objects via attributes whose values are XGL object pointers.

Table 4-1 API User Object and Used Object Relationships

User Object	Linking Attribute	Used Object Class
XglContext2d	XGL_CTX_DEVICE	XglRasterMem XglRasterWin
	XGL_CTX_GLOBAL_MODEL_TRANS	XglTransform
	XGL_CTX_LOCAL_MODEL_TRANS	XglTransform
	XGL_CTX_MODEL_TRANS (Read-only)	XglTransform
	XGL_CTX_VIEW_TRANS	XglTransform
	XGL_MC_TO_DC_TRANS (Read-only)	XglTransform
	XGL_CTX_LINE_PATTERN	XglLinePattern
	XGL_CTX_EDGE_PATTERN	XglLinePattern
	XGL_CTX_MARKER	XglMarker
	XGL_CTX_RASTER_FPAT	XglRasterMem
	XGL_CTX_SURF_FRONT_FPAT	XglRasterMem

Table 4-1 API User Object and Used Object Relationships (Continued)

User Object	Linking Attribute	Used Object Class
	XGL_CTX_SFONT_0	XglSfont
	XGL_CTX_SFONT_1	XglSfont
	XGL_CTX_SFONT_2	XglSfont
	XGL_CTX_SFONT_3	XglSfont
XglContext3d	All the above and:	
	XGL_3D_CTX_NORMAL_TRANS (Read-only)	XglTransform (3D)
	XGL_3D_CTX_SURF_BACK_FPAT	XglRasterMem
	XGL_3D_CTX_LIGHTS	XglLight
	XGL_3D_CTX_SURF_FRONT_TMAP XGL_3D_CX_SURF_BACK_TMAP	XglTextureMap
	XGL_3D_CTX_SURF_FRONT_DMAP XGL_3D_CX_SURF_BACK_DMAP	XglDmapTexture
XglDevice	XGL_DEV_COLOR_MAP	XglCmap
	XGL_DEV_CONTEXTS	XglContext
XglRasterMem	Same attributes as XglDevice	
XglRasterWin	Same attributes as XglDevice	

Architecture of Object Relationships

The relationship between API objects is implemented with an object registration and message passing mechanism. This mechanism is defined in the XglObject class and is inherited by XglObject's subclasses. It is designed to do the following:

- Inform interested user objects of changes in used objects.

  When a used object is changed, it might need to communicate its changes to some of its users. For example, if a Device is attached to a Context and the Device color map is changed, the Device needs to warn the Context that the color map has changed. In some cases, however, users might not be concerned about changes and, therefore, might not want to be informed.

- Delay the destruction of a used object until it no longer has users.

  When an object is asked to destroy itself (by `xgl_object_destroy()`), it must only destroy itself if it does not have any other object referencing it. If it does have another object referencing it, it must postpone its destruction until it no longer has users. For example, if a Transform object is attached to the Context object and the API operator `xgl_object_destroy()` is invoked to destroy the Transform, the Transform must not destroy itself immediately, since it is still referenced by the Context object. It can only destroy itself when the Context attribute referencing the Transform is reset to another Transform.

The basic data structure of the object registration and message passing mechanism is the user list.

Object Registration: The User List

An API object stores information on associated objects in its user list. The user list is a linked list that stores a pointer to the user object, a Boolean value indicating whether the user object wants to be notified of changes, and a reference count on a per object basis.

The reference count records how many times a user object has registered itself with the used object. For example, an application program can associate the same Line Pattern object with a Context object twice, once to set the pattern of lines and once to set the pattern of surface edges. If the application does this, the Line Pattern object will register the Context object as a user two times, and the reference count of the Line Pattern object's user list will be incremented accordingly. If the pattern for lines is then changed to a different pattern, the Context will remove itself as a user of the original pattern object for line patterns and register itself as a user of the new pattern object. When this occurs, the original pattern object decrements its reference count for the Context object by one. The user list is illustrated in Figure 4-6.

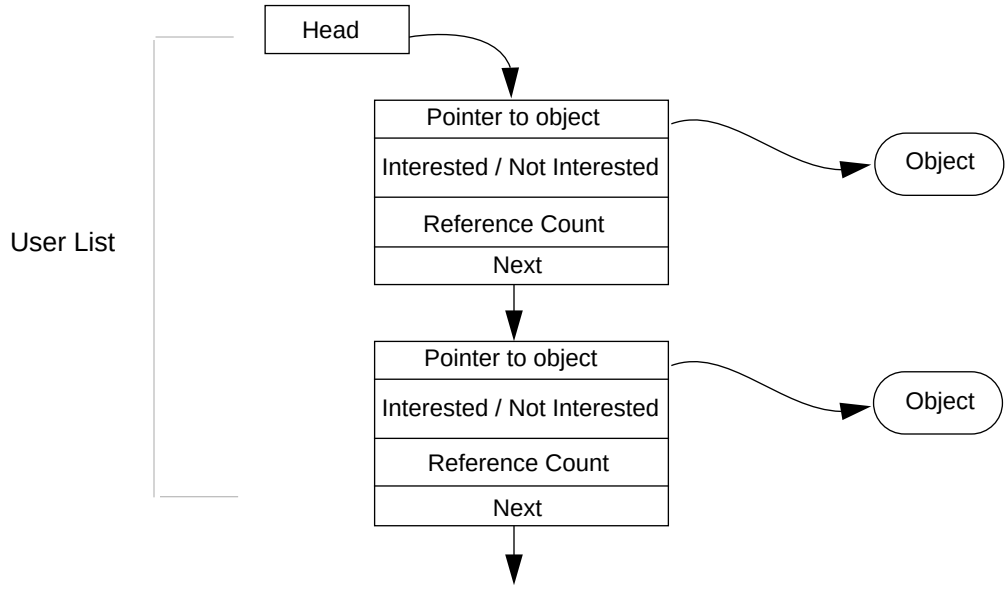


Figure 4-6 User List

How the User List Works

An object registers itself with another object by invoking the `addUser()` member function of the XglObject class and specifying whether it is interested in receiving a message from the object if a change occurs in the object's state. A call to `addUser()` is shown below:

```
void addUser(XglObject* obj,Xgl_boolean notify_interest =FALSE);
```

`addUser()` determines whether the registering object is already registered in the used object's user list. If the registering object is not in the user list, `addUser()` adds a new node to the user list. If the `notify_interest` variable is set to `TRUE`, `addUser()` registers the object as interested in being notified if the used object changes, and, if the registering object has registered

itself before, the used object increments the reference count for that user and OR's the current `notify_interest` flag with the already existing flag. The `addUser()` function is defined in `Object.cc` as:

```
void XglObject::addUser(XglObject* obj, Xgl_boolean notify_interest)
{
    XglUser* user = userList;

    while (user) {
        if (user->object == obj) {
            user->refCount++;
            user->interest = user->interest || notify_interest;
            return;
        }
        user = user->next;
    }

    // Create the user
    user = new XglUser(obj,notify_interest);
    user->next = userList;
    userList = user;
}
```

When a user object no longer uses another object, it invokes `removeUser()` to remove it from the used object's user list. `removeUser()` traverses the used object's user list, locates the node for the user object, and decrements the reference count field for that object node by one. If the reference count is zero, the node is deleted from the user list. `removeUser()` is a member function of the XglObject class and is declared in `XglObject.h`.

As a side effect, `removeUser()` deletes the used object if the API operator `xgl_object_destroy()` has been invoked to delete the object. However, the used object will not destroy itself if it has any users but will simply note that it was asked to destroy itself. Thus, if a used object has been asked to destroy itself and the used object's only user asks the used object to remove the user from the used object's user list, the used object traverses its list, decrements the reference count, deletes the node, and then destroys itself.

`removeUser()` is defined in `Object.cc` as:

```
void XglObject::removeUser(XglObject* obj)
{
    XglUser* user, *prev_user;

    if (userList) {
        for (user=userList, prev_user=user; user;
            prev_user=user, user=user->next){
            if (user->object == obj) {
                if (--(user->refCount)==0) {
                    if (user == userList) //object is in first node
                        userList = user->next;
                    else
                        prev_user->next = user->next;
                delete user;
                if (destroyed && (userList == NULL))
                     delete this;
            }
            return;
        }
    }
  }
}
```

Message Passing

Whenever an API object is changed, some or all of its users might want to be notified of the changes. For example, if a Window Raster Device object is attached to many Context objects, all of which have expressed an interest in being notified of changes, and the dimensions of the window change, the Device object must notify its interested users that its state has changed.

It is the responsibility of the user object to inform the used object that it is using the used object and that it is interested in being notified of any changes. It is the responsibility of the used object to notify interested users whenever a state change occurs. These operations are handled by the XglObject member functions `send()` and `receive()`.

How Message Passing Works

A used object sends a message to interested users by invoking the XglObject member function `send()`. This function traverses an object's user list and sends a message to each of the object's interested users. `send()` specifies who the sender is and contains information on what aspect of the object's state has changed. The function is defined as follows:

```
void XglObject::send(const XglMsg &msg)
{
    XglUser* user = userList;

    while (user) {
        if (user->interest) {
            user->object->receive(this,msg);
        }
        user = user->next;
    }
}
```

A user object processes the local impact of changes of a used object with the `receive()` function. `receive()` has two arguments: a pointer to the object that is sending the message and the message itself. When an object, such as the Context, receives a message, it determines which of the objects it is currently using sent the message, and it does processing based on the nature of the message. When the Context has processed the message, it passes the message to its parent class, and this class, in turn, passes the message to its parent class, until the message reaches the top of the DI hierarchy. The message is passed upward because the attributes associated with the message might be defined in any of the object's parent classes.

`receive()` is a virtual function in `XglObject.h` and is invoked as follows:

```
virtual void receive(XglObject* obj, const XglMsg &msg);
```

`receive()` can be overridden by each API object to handle the specific processing for its own attributes. Only a given class or subclass can do the specific processing triggered by an incoming message from an object currently associated with an object of this class.

Destroying Objects and Closing XGL

All API objects are destroyed through the System State object. When the application program calls `xgl_object_destroy()` to destroy an object, the System State object searches the appropriate API object list, removes the object from the list, and then calls the destroy function for the API objects. This destroy function is also used to destroy all existing API objects during `xgl_close()`.

Destroying the Device Object

An application call to `xgl_object_destroy()` with a Device handle as the input parameter destroys the Device object if it is not used by any Context object. When the Device object is asked to destroy itself, the following events occur:

1. The System State object removes the Device object from its Device object list and destroys the Device object if it is not referenced by any users.
2. The Device object determines whether if has any users. If it does not have any users, it notifies all the objects that it is currently using that it will no longer use them. Then it destroys itself. In the process, it also destroys its device-dependent part, XglDpDev. If it has users, it notes that it was asked to destroy itself.

Destroying the Context Object

An application call to `xgl_object_destroy()` with a Context handle as the input parameter destroys the Context object. When a Context is destroyed, the following events occur:

1. The System State object removes the Context object from the list of objects it has created.
2. The XglSysState asks the Context object to destroy itself. During this process, the Context object determines whether it has any users.
   a. If the Context has users, it notes that it was asked to destroy itself.
   b. If the Context does not have users, it does the following:
      i. It notifies all the objects that it is currently using that it will no longer use them.

ii. It destroys the software pipeline object.

iii. It destroys itself.

Closing XGL

When XGL is closed, the System State object deletes all the objects it created in the same process that is used in an `xgl_object_destroy()` call. The pipeline XglDpDev object is destroyed as part of the Device object destruction, and the XglDpCtx object is destroyed during the corresponding Device destruction or Context destruction. The System State object then deletes itself.

During `xgl_close()`, the Global State object is destroyed, and the destructor of the XglDpLib object is called. The XglDpLib destructor is explicitly invoked using the handle to the object. It is the responsibility of the XglDpLib object to destroy the XglDpMgr object(s) for the pipeline, since there may be one or more XglDpMgr objects for a particular pipeline.

The Global State object executes `dlclose()` to remove the reference to the pipeline shared object from XGL's process space.

Rendering and Handling State Changes 5

This chapter describes the processes of rendering and handling Context state changes. It includes information on the following topics:

- Steps in the rendering process
- Architecture of the mechanisms for storing state changes and passing information about the changes to the device pipeline
- Architecture of backing store

Goals of the Rendering Architecture

The XGL architecture provides flexibility in the interactions between the device pipeline and the software pipeline at rendering time. The device pipeline can choose to accelerate some rendering tasks and to fall back on the XGL software pipeline for other rendering tasks. This dynamic switching between the device pipeline and the software pipeline allows the device pipeline implementor to tailor a port for a specific device.

The goals for the rendering architecture were to:

- Allow the device pipeline LI-1 rendering function to make the decision that it cannot render a primitive and call the software pipeline for LI-1 or LI-2 processing.
- Allow the software pipeline at the LI-1 layer to call the device pipeline at the LI-2 layer.

- Allow the device pipeline to call the equivalent software pipeline function within a rendering call if the device pipeline cannot completely accelerate the primitive.
- Allow the software pipeline to call a different rendering function at the same level without knowing whether the function is implemented in the device pipeline. For example, if the device pipeline has not implemented stroke text, it can call the software pipeline's LI-1 stroke text function; the software pipeline's stroke text function may then tesselate the text into lines and call back the device pipeline LI-1 polyline function to see if it can render the partially processed geometry data.

Basic information on the steps of the rendering process is provided in the following section.

How Rendering Works

An application call to render a primitive is mapped by the device-independent code to a call to the device pipeline. This mapping takes place in the Context wrapper. For example, the API multirectangle operator is:

```
xgl_multirectangle(ctx,rect_list);
```

The wrapper code maps the C function call to the C++ internal code and forwards the call and the application data to the device pipeline. To do this, the wrapper gets a pointer to the current pipeline and calls the pipeline's version of `xgl_multirectangle()`. The wrapper locates the pipeline versions of the XGL primitives in an array of function pointers, which is called the `opsVec` array. The sample code below shows the `currOpsVec` pointer pointing to the multirectangle entry of the `opsVec` array.

```
void xgl_multirectangle(Xgl_ctx ctx, Xgl_rect_list* rect_list)
{
    // Get pointer to device pipeline
    XglDpCtx* dp = ((XglCtxObject*)ctx)->getDp();

    (dp->*((void(XglDpCtx::*)(Xgl_rect_list*))
           (dp->currOpsVec[XGLI_LI1_MULTIRECTANGLE])
          ))(rect_list);
}
```

Communciation Between Device-Independent XGL and the Device Pipeline

Rendering calls and information on attribute changes are passed from the device-independent code to the device pipeline through the opsVec array. The opsVec array is a dynamic array of function pointers to device pipeline or software pipeline renderers. It serves as the point of communication between the device-independent code and the device pipeline code. The opsVec array is designed to minimize the overhead in the device-independent code during each graphics primitive call. When the application calls a primitive or makes an attribute change, the pipeline is notified of this immediately through the opsVec array.

The opsVec array is allocated by the base class XglDpCtx and filled in with pointers to default base class functions that point to software pipeline functions. For example, a portion of the 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_LI1_NURBS_CURVE] = XGLI_OPS(XglDpCtx3d::li1NurbsCurve);
opsVec[XGLI_LI1_MULTIRECTANGLE] = XGLI_OPS(XglDpCtx3d::li1MultiRectangle);
opsVec[XGLI_LI1_MULTIARC] = XGLI_OPS(XglDpCtx3d::li1MultiArc);
opsVec[XGLI_LI1_MULTICIRCLE] = XGLI_OPS(XglDpCtx3d::li1MultiCircle);
opsVec[XGLI_LI1_POLYGON] = XGLI_OPS(XglDpCtx3d::li1Polygon);
.....
```

When the device pipeline is instantiated, it creates a version of the XglDpCtx object for its own pipeline. This object contains a set of opsVec array pointers specific to the device. In its version of the array, the device pipeline will override some or all of the entries in the opsVec array to install pointers to its renderers. If the device pipeline does not support a primitive, it can let the default function call the software pipeline for rendering. If a device pipeline can accelerate a primitive, it can override the default array entry with a function pointer to its primitive. For example, the GX pipeline installs the

following opsVec entries in its XglDpCtx3dGx class. For the remaining primitives, the array will inherit the default entries pointing to the software pipeline when the object is instantiated.

```
opsVec[XGLI_LI1_MULTIMARKER] = XGLI_OPS(XglDpCtx3dGx::li1MultiMarker);
opsVec[XGLI_LI1_MULTIPOLYLINE] = XGLI_OPS(XglDpCtx3dGx::li1MultiPolyline);
opsVec[XGLI_LI1_MULTI_SIMPLE_POLYGON] =
                          XGLI_OPS(XglDpCtx3dGx::li1MultiSimplePolygon);
opsVec[XGLI_LI1_TRIANGLE_STRIP] = XGLI_OPS(XglDpCtx3dGx::li1TriangleStrip);
```

Figure 5-1 shows a device pipeline that implements polyline and polygon renderers but points to the software pipeline for text rendering.

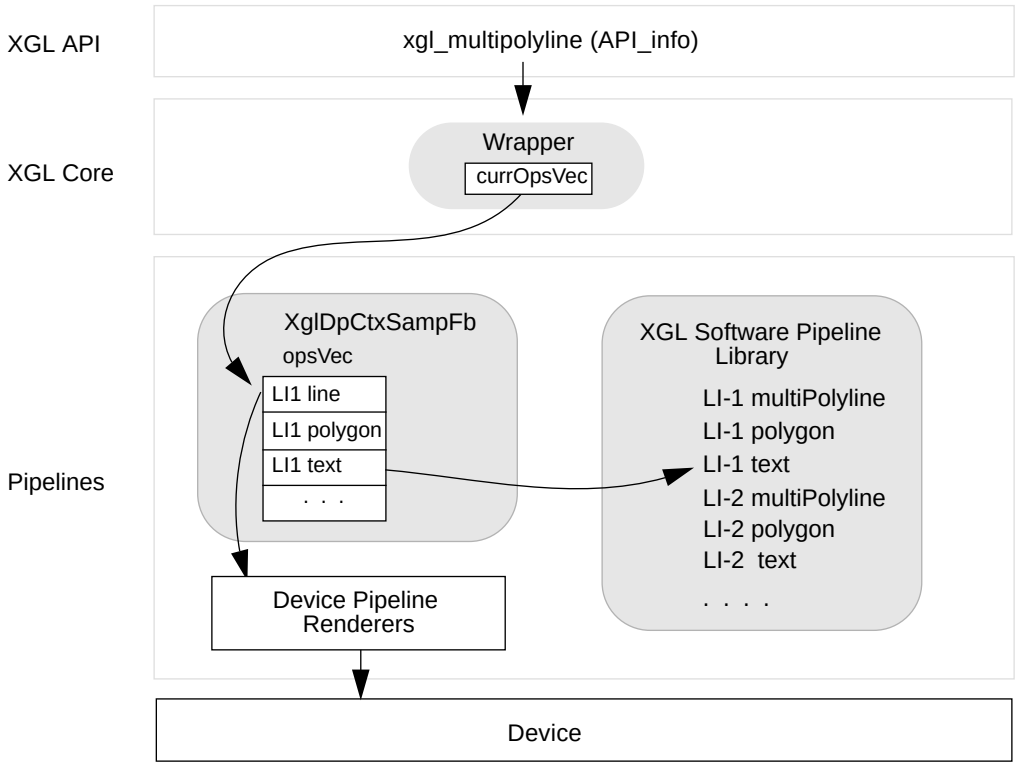


Figure 5-1 Rendering Through the opsVec Array

Device Pipeline Options for Rendering

When a device pipeline is loaded, it initializes its XglDpCtx object to assign pointers to its renderers. The opsVec array architecture allows the device pipeline to create a set of renderers that are tuned for performance. For primitives that the pipeline wants to override but that do not need to have optimal performance, the pipeline might create a single function that handles error checking and attribute setting. For performance-critical primitives, the device pipeline may want to create a set of functions including a generic or "slow" function that performs error and attribute checking and one or more functions that are optimized to handle particular sets of attribute values. These "fast" renderers are used when a primitive is called repeatedly without attribute changes. Depending on the sequence of primitive-attribute calls that preceded a particular call, the pipeline can determine which renderer it should set in the opsVec array.

Thus, at rendering time, the device pipeline has several options:

- If the device pipeline has not implemented a rendering function, it need not reset the opsVec entry but can inherit the pointer to the default function, which calls the software pipeline.
- The device pipeline can accelerate the primitive by installing a pointer to a single device pipeline renderer that handles error checking and attribute checking.
- The pipeline can design a set of functions for primitives whose performance is critical. In this case, one renderer may handle generic attribute checking and call another renderer that simply sends data to the hardware given certain attribute values. When attributes change, the pipeline will again call the generic renderer to check attribute changes; when the slow processing of attribute checking is complete, the generic renderer can call the optimized renderer.

Even when the device pipeline has implemented a primitive, it may need to call the software pipeline for assistance at times. The device pipeline calls the software pipeline directly. It can do so in the following cases:

- If the device pipeline doesn't support the current combination of attribute values, it can call the software pipeline without processing the primitive call.

- If the device pipeline doesn't support the combination of primitive argument values, it must process the primitive call, but it can call the software pipeline before calling a renderer.
- In some cases, the device pipeline may begin processing the data and decide that it cannot render all the data. For example, the device pipeline may request the software pipeline to:
  - Render a primitive, as in the case of a clipped polygon that the device pipeline cannot handle in a `xgl_polygon()` call.
  - Render a subset of the primitive data, as in the case of a polygon that the device pipeline cannot handle in a `xgl_multi_simple_polygon()` call.

  In this case, the pipeline must call the software pipeline from inside the renderer.

Note that the device pipeline can reset the `opsVec` array entries whenever it needs to. Although the array is defined in the device-independent code, the DI will never set `opsVec` entries, and there are no restrictions on when the device pipeline can change its renderers. For information on how the device pipeline sets pointers to its functions in the `opsVec` array, see the *XGL Device Pipeline Porting Guide.*

More on the Rendering Architecture

Because the `opsVec` array was designed to provide the device pipeline with optimal performance, a way for the device-independent code to handle low performance functionality, such as backing store and error checking, was needed. In particular, the device-independent code needed a way to handle rendering to backing store that would not impact rendering to the primary device. To meet this need, an array of wrappers to the `opsVec` function pointers was designed. This array, called `opsVecGen`, is a static array that points to a set of general purpose functions that handle error detection and backing store, render through the `opsVec` array, and then check the deferral mode. Note that the device pipeline cannot change the function pointers in the `opsVecGen` array as it can in the `opsVec` array.

The device-independent code maintains a pointer to specify whether it is rendering to the `opsVec` array or the `opsVecGen` array. This pointer is called `currOpsVec` and is defined in `DpCtx.h` along with the two arrays. If the application sets the backing store on a device using the attribute `XGL_WIN_RAS_BACKING_STORE`, the device-independent wrapper will set the

currOpsVec pointer to point to the opsVecGen array instead of the opsVec array. Any attribute changes are sent to both the primary device and the backing store device to keep them synchronized.

Figure 5-2 shows the opsVecGen architecture. For more information on backing store, see page 95.

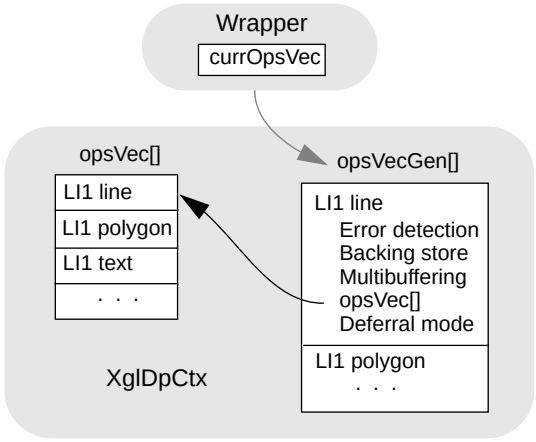


Figure 5-2 opsVecGen Architecture

Context State Changes

The device pipeline is notified of Context state changes when the changes occur. The application can cause Context state to change in two ways:

- It can change a Context attribute directly using xgl_object_set().
- It can change another object, which indirectly causes a Context attribute to change.

In addition, view model or Transform changes cause changes in a set of items derived from the Context's view model attributes.

Information on attribute changes is passed directly to the device pipelines through the opsVec array. The array has an entry for attribute changes resulting from xgl_object_set() and another entry for attribute changes resulting from messages between objects. The array is designed to minimize

the function call overhead for notifying the device pipeline of attribute changes. It also allows the device pipeline to act only on the attributes that it is concerned with and ignore all other attributes.

The sections that follow discuss how the `opsVec` array passes information on attribute changes to the device pipeline and how the pipeline can get new attribute values when changes occur.

State Changes From Attribute Setting

Context state changes can result from an application `xgl_object_set()` call with the *obj* parameter of `ctx`. An object set call on the Context:

```
xgl_object_set(ctx, Xgl_CTX_ATTR_NAME, value, NULL);
```

maps to the following C++ call:

```
ctx->setAttrName(value);
```

When an attribute set operation changes a Context attribute value, the Context wrapper notifies the Context object of the change. The Context updates its cached attribute value and builds an array of attribute changes. This array lists attributes that have changed but does not include the new attribute values.

When the entire list of attributes is processed, the wrapper gets a pointer to the current device pipeline from the Context object, gets the list of changed attribute types from the Context, and sends the list of changes to the pipeline through the object set entry of the `opsVec` array.

The device pipeline must provide an `objectSet()` function to handle attribute changes and install a pointer to this function in its `opsVec` array. The `objectSet()` function will check the attribute type of the attributes that the pipeline is interested in, retrieve the attribute values from the Context, and update the hardware state. One way of handling changes is to get the new attribute value from the Context object immediately, as in the following example code:

```
void XglDpCtx2dGx::objectSet(const Xgl_attribute *att_type)
{
    Xgl_usgn32 change_renderer = 0;

    for (;*att_type;att_type++) {
        switch(*att_type) {

        case XGL_CTX_DEVICE:
           // Update all context attributes.
           // (ctx->getAttrTypeListAll()); // DI utility list.
           objectSet((const Xgl_attribute*) attr2dTypeListStatic);
           break;
        case XGL_CTX_BACKGROUND_COLOR:
           // Update backgroundColor (background cached color)
            dpDev->cMap->getColorMapper()(dpDev->cMap,
                          &(Xgl_color&)ctx->getBackgroundColor(),
                          &backgroundColor, TRUE);
           // update hardware
           break;
       case XGL_CTX_SURF_FRONT_FILL_STYLE:
           surfIndex.fld.fstyle = ctx->getSurfFrontFillStyle();
           // update hardware
           break;
       ...
       }
    }
}
```

As an alternative, the pipeline can choose to simply note that an attribute was changed and wait until a later time to update the hardware context, perhaps updating the hardware context at rendering time. At that time, the device pipeline will check the XGL Context for the new attribute values and update the hardware. The following sample code shows a pipeline `objectSet()` routine checking for attribute changes.

```
void XglDpCtx3dSampFb::objectSet(const Xgl_attribute *attr_list)
{
    Xgl_attributeatt_type;

    while (att_type = *attr_list) {
       switch(att_type) {
       case XGL_CTX_ATEXT_CHAR_SLANT_ANGLE:
           update_needed = TRUE;
           break;
       case XGL_CTX_DEVICE:
```

```
            objectSet(ctx->getAttrTypeListAll());
            break;
        case XGL_CTX_EDGE_COLOR:
            update_needed = TRUE;
            break;
        ...
        }
    }
}
```

Note that if the Device is changed, the device pipeline is responsible for updating its entire attribute cache.

State Changes From XGL Object Message Passing

Context state changes can result from changes to objects used by the Context. For example, the Context object registers itself as a user of the Device, the Line Pattern, and the Stroke Font objects, and notes that it wants to be notified when changes to these objects occur. When an object the Context is using changes, the object updates its data and sends a message to the Context. The Context receives the message and reacts accordingly. Objects register interest in other objects through the user list. For more information on the user list, see page 70.

Objects can change in two ways:

- The application can change the object that it is using; in other words, it can set a different Line Pattern object on the Context. This change is handled by the object set mechanism, and the device pipeline is informed of this change through the object set entry of the `opsVec` array. Note that a Color Map set on a Raster is handled by message passing; see page 94 for more information on Color Map changes.
- The object's data can change, as would be the case if the application changed the line pattern data in an existing Line Pattern object. This change is handled by the message receive mechanism.

In the message receive mechanism, the object sends a message to its users informing them of the change. For example, if a Line Pattern object changes, it sends the Context a message informing it that the Line Pattern object has changed. The Context receives this message in its `receive()` function and updates its attribute data. The Context then forwards the message to the device pipeline through the message receive entry of the `opsVec` array.

To handle forwarded messages, the device pipeline must provide a `messageReceive()` function and install a pointer to this function in the `XGLI_LI_MSG_RCV` entry of its `opsVec` array. The pipeline's `messageReceive()` function will check the object type and respond to the message, as is shown in the following code fragment.

```
void XglDpCtx{2,3}dGx::messageReceive(XglObject* obj,
                                      const XglMsg& msg)
{
    switch (obj->getObjType()) {
      case XGL_2D_CTX:
      case XGL_3D_CTX:
        if (msg.flag & (XGLI_MSG_VIEW_COORD_SYS |
                        XGLI_MSG_VIEW_CTX_ATTR)) {
          transformsChanged = TRUE;
          // Set generic renderers.
        }
        break;

      case XGL_WIN_RAS:
        if (obj == device) {
            if (msg.flag & XGLI_MSG_DEV_COLOR) {
            // Update cached colors and plane mask changes.
        }
       break;

      case XGL_LPAT:
        if (obj == ctx->getLinePattern() ||
              obj == ctx->getEdgePattern()) {
           objectSet((const Xgl_attribute*) attrTypeList);
       }
        break;
      case XGL_TRANS:
        if (obj == (XglObject*)ctx->getGlobalModelTrans() ||
            obj == (XglObject*)ctx->getLocalModelTrans() ||
            obj == (XglObject*)ctx->getViewTrans()) {

          transformsChanged = TRUE;
          // Set generic renderers.
       }
       break;
       ...
   }
}
```

For more specific information on the messages the device pipeline needs to respond to, see the *XGL Device Pipeline Porting Guide.*

View Model Derived Data

XGL defines a conceptual view model consisting of a number of coordinate systems where an application can specify certain operations. XGL provides a facility named *view model derived data* to assist pipelines with implementation of the view model operations.

Derived data maintains a cache of items derived from a Context's view model attributes. The derived items include Transforms for mapping geometry between coordinate systems as well as items in various coordinate systems such as the view clip bounds, lights, eye positions or eye vectors, model clip planes, and depth cue reference planes. For example, derived data calculates the VDC-to-DC Transform from the Context attributes for the VDC map, VDC orientation, VDC window, DC viewport, and jitter offset, and the Device attribute for DC orientation. Thus, a derived item can depend on only API attributes, on only derived items, or on a combination of both.

The design goals of derived data are to:

1. Support geometry entering LI-1 from other coordinate systems (in addition to Model Coordinates) with a simple interface for pipelines. The pipeline can set the current coordinate system when one primitive invokes another. For example, `li1AnnotationText()` can decompose the text into polylines in VDC and then invoke `li1MultiPolyline()`. Derived data keeps track of the current coordinate system for the pipeline.

2. Provide a fast test to inform a pipeline of changes to derived items of concern so that the pipeline can minimize data transfer to devices that retain state. A pipeline can express concern about changes to a specified set of items. This allows pipelines to filter out irrelevant changes. This is useful because derived data consists of a large number of items and pipelines are typically interested in only a few of the items.

3. Defer calculation of a derived item until a pipeline requests that item. The pipelines are not notified of derived data changes immediately but when they need them at rendering time.

Components of the Derived Data Mechanism

The central object of the derived data mechanism is the view cache object. The view cache consists of derived items and functions for deferred evaluation of the items. Each Context has a pointer to its own view cache, which maintains the derived items specific to that Context. A view cache object is created at Context creation time by a constructor in the Context object.

The Context constructor creates a set of Transform objects that represent its default transformations (Local Model Transform, etc.) and creates the view cache object. The view cache constructor creates internal Transforms that are needed by derived data. The view cache constructor also creates a set of view group configuration objects, each of which represents a coordinate system from which geometry can enter an LI-1 primitive.

The view group interface object is the pipeline's interface to the view model derived data. It informs a pipeline when derived items have changed as a result of either the application changing a view model attribute or a pipeline changing the coordinate system from which geometry enters the next LI-1 primitive. The view group interface constructor is called in the XglPipeCtx{2,3}d classes, and the object is created when the pipeline XglDpCtx object is created.

Each pipeline has a pointer to a view group interface object. The view group interface has functions for creating and destroying view concern objects. A view concern object is a description of all the derived items about which a pipeline is concerned. A pipeline may create as many view concern objects as it needs. For example, it can create one view concern object for stroke primitives and another for surface primitives.

Figure 5-3 on page 90 illustrates the set of objects in the derived data mechamism. This example shows a 3D Context with one view concern object for stroke primitives and another for surface primitives. Note that for 3D Contexts, there are five view group configuration objects.

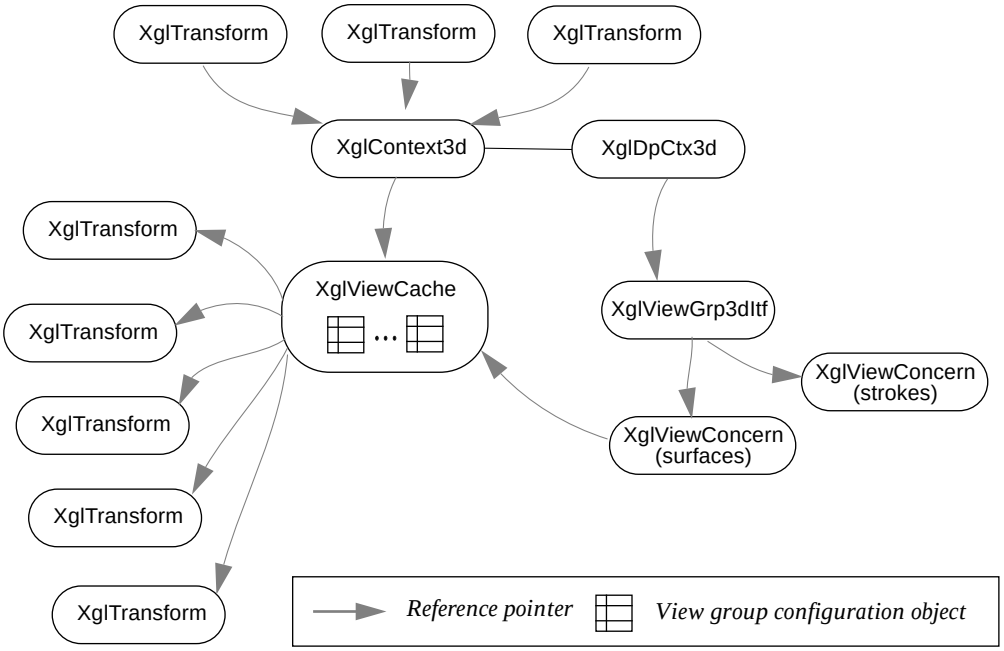


Figure 5-3 Derived Data Mechanism

How Derived Data Gets Information From the Context

As mentioned on page 83, there are two ways that Context state can change: the application can directly set a Context attribute, or the application can change an object used by the Context, causing the object to send a message to the Context about the change. Either of these two types of changes can cause derived data to change.

The Context has a number of attributes that the application can set, among which are the view model attributes. For example, if the application changes the VDC window using `xgl_object_set()` to set `XGL_CTX_VDC_WINDOW`, a number of derived items that are dependent on the value of the VDC window become invalid. But, because the changes are evaluated lazily, the derived data items are not recalculated until the pipeline requests the items.

If the member datum in the Context for the VDC window is changed, the Context informs the view cache object that the application changed the VDC window by calling a function in the view cache object. This function invalidates the derived items that depend on the VDC window. A set of bits records which derived items changed.

Some Context attributes that are objects, such as Lights, Transforms, or the Device, can change, causing derived data items to change. For example, if an application gets the Local Model Transform and changes it, the Transform sends a message to each Context that uses the Transform. If the Context is interested in the change to the Transform, it will call a view cache function to invalidate derived items that depend on the Local Model Transform. This process is similar for changes to Lights and for Device resizes.

Handling Derived Data Changes

Derived items can change when the application changes a view model attribute or a pipeline changes the current coordinate system. Each type of event causes a message to be sent to the device pipeline at the time of the event; notification is not deferred. The message types are `XGLI_MSG_VIEW_CTX_ATTR` and `XGLI_MSG_VIEW_COORD_SYS` for Context attribute changes and current coordinate system changes, respectively.

Messages of the two types above give advance warning that the next primitive may need to get derived items. A pipeline may choose to deal with the messages simply by setting its own flag at the time of the notification, then deferring action until the next primitive when it would need to interrogate the composite at the next level.

A pipeline determines that a Context attribute or derived item has changed by checking the flag that the pipeline sets upon receiving a message of the types `XGLI_MSG_VIEW_CTX_ATTR` or `XGLI_MSG_VIEW_COORD_SYS`. If the flag is set, the pipeline determines whether any derived items have changed by calling the `changedComposite()` function in the view group interface object, as in `viewGrpItf->changedComposite(surfConcern)`. The composite is a record of the state changes for all derived items. If the `changedComposite()` function returns `TRUE`, the pipeline must check the change functions for individual items to determine which ones changed. For more information on using the view group interface object's change functions, see the *XGL Device Pipeline Porting Guide.*

Changes in Context Stroke Attributes

The Context object includes a set of objects that contain attribute information used when the software pipeline draws primitives as lines. For example, if a device pipeline does not implement text, the software pipeline will render the text. The software pipeline will convert the text to polylines and call the device pipeline polyline function to render the lines. However, in this case, the device pipeline polyline function will want to use the text attributes rather than the line attributes when rendering the lines. This is handled transparently for the device pipeline by the stroke group objects.

Lines, markers, text, edges, and front and back hollow polygons are considered to be *stroke types*. Since the same set of attributes applies to each of the stroke types, the Context object maintains a stroke group object for each of the stroke types. Multiplexing primitives on the `MultiPolyline()` primitive is shown in Figure 5-4.

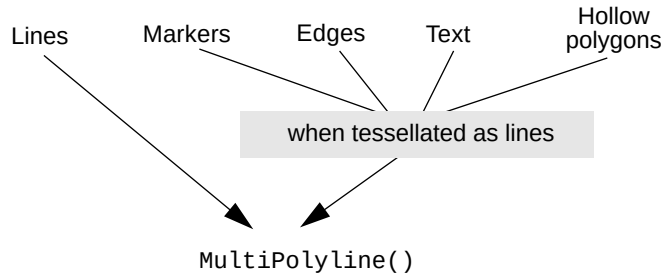


Figure 5-4 Multiplexing Primitives on MultiPolyline()

Note – Stroke groups are used only by the `MultiPolyline()` primitive. Other stroke primitives, such as `MultiMarker()`, do not use stroke groups.

Stroke Group Objects

The stroke group object for each stroke type contains the values for the multipolyline-specific attributes (line color, line width, etc.). When an attribute set occurs on a Context, the Context updates the attribute's value, and, if the set affects a stroke group, the Context also updates the stroke group attribute for that stroke type.

The XglStrokeGroup object provides interfaces for the pipeline to retrieve the value of stroke attributes. For example, if a pipeline needs to know the value of the line color, it invokes the stroke groups's `getColor()` function. The XglStrokeGroup3d object adds 3D-specific attributes to the stroke object. The stroke group objects are associated with the Context object and are created at Context creation time. A conceptual sketch of the relationship between the Context object and the stroke group objects is shown in Figure 5-5.

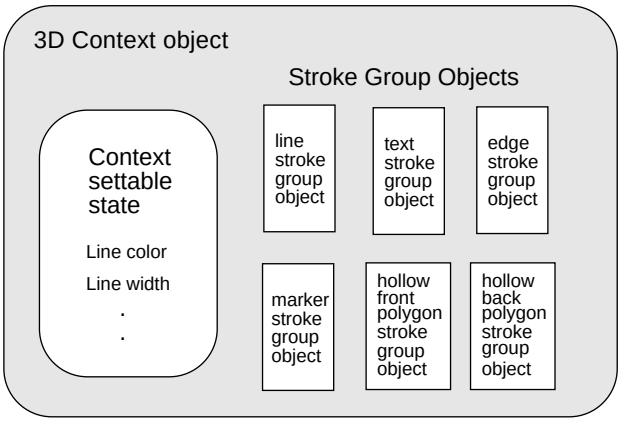


Figure 5-5 Stroke Group Objects in the 3D Context Object

Rendering Using Stroke Attributes

Each stroke group object contains the attribute values for a particular stroke type. To indicate which stroke group will be used for rendering, the Context object provides a current stroke pointer, which points to one of the stroke group objects. When geometry is rendered as lines, the stroke pointer is set to the appropriate stroke group and `MultiPolyline()` renders using that stroke group.

For example, if a device pipeline does not implement text, the software pipeline will tesselate the text into polylines and call the device pipeline's `MultiPolyline()` function. However, before this happens, the software pipeline assigns the current stroke group to the text stroke group. This generates an `objectSet()` call to the device pipeline indicating which attributes need to be loaded from the current stroke group. By the time the device pipeline `MultiPolyline()` function is called, all the correct stroke attributes will already have been processed.

5

Device State Changes

The state of a Device object can change in the following general ways:

1. Device changes can be color or pixel mapping related. The application can modify the Color Map object associated with the Device, or it can set a new Color Map on the Device.
2. Device changes can result from changes to the dimensions of the raster. In this case, Context state is affected since Window Raster size changes cause derived data changes.
3. Device changes can result from switching buffers during multibuffering.

The pipeline XglDpCtx object and XglDpDev object are each informed of Device state changes, as follows:

- The device pipeline XglDpCtx object is notified of Device changes via the Context object through message passing. This enables the XglDpCtx to update hardware state. For example, the device-independent Device object sends a message to the Context about a color map change in these lines:

  ```
  XglMsg msg;
  msg.flag = XGLI_MSG_DEV_COLOR;
  send(msg);
  ```

  The Context receives the message and forwards it to the XglDpCtx through the `opsVec` array.

- The device pipeline XglDpDev object is notified of a change by a direct call from the device-independent code. This allows the pipeline to make device-specific changes. For example, this call informs XglDpDev of a color map change:

  ```
  ((XglDpDevWinRas*)dpDev)->setCmap(cmap);
  ```

See the *XGL Device Pipeline Porting Guide* for more information on the message passing mechanism and on the XglDpDev functions that the device-independent Device object uses to inform the pipeline of device state changes.

Rendering Into Backing Store

A backing store consists of a piece of off-screen memory that reflects the content of the display buffer and that is used by the server to restore previously obscured areas of the display during an expose event. When backing store is enabled, XGL renders a primitive to both the visible and covered portions of the window when the window is clipped. Normally, XGL renders the covered area to backing store using the inverse clip list and does not render the exposed area to backing store. However, if the inverse clip list becomes complex, it is more efficient to render to the entire backing store, and XGL will, at its option, delete the inverse clip list.

In general, the XGL core is responsible for handling backing store operations. The device pipeline need only implement a small set of device-dependent functions in the pipeline to provide support in certain cases. However, the device pipeline has the option of handling backing store itself. For example, the PEXlib pipeline can rely on the server to handle backing store operations, so it does not need the XGL core to provide backing store support.

Architecture of Backing Store

An application that wants to use backing store first requests it through the server and then through XGL. When the server is asked to provide backing store, it tries to allocate backing store memory from either system memory or from a hardware cache. It is up to the server to decide whether to provide the backing store support and from which area the backing store is allocated.

When XGL is asked to enable backing store, it determines whether the server granted the request for backing store and whether the window is single buffered (since XGL cannot render to backing store if double buffering is enabled). If backing store is enabled, and the window is single buffered, XGL then requests the handle to the allocated memory through the XglDrawable.

To render into the backing store area, XGL creates a set of shadow objects that reflect the contents of the parent objects. For example, each Context and Device pair is associated by a device pipeline-context object. Thus, if there is a shadow device, there is a parent XglDpCtx, which renders the visible part of the screen, and a shadow XglDpCtx, which renders to the backing store. The shadow objects are:

- Shadow device pipeline-context object

- Shadow raster object, which can be either a Memory Raster or a Window Raster
- Shadow device pipeline device object

Figure 5-6 shows the shadow devices that are created when backing store is enabled.

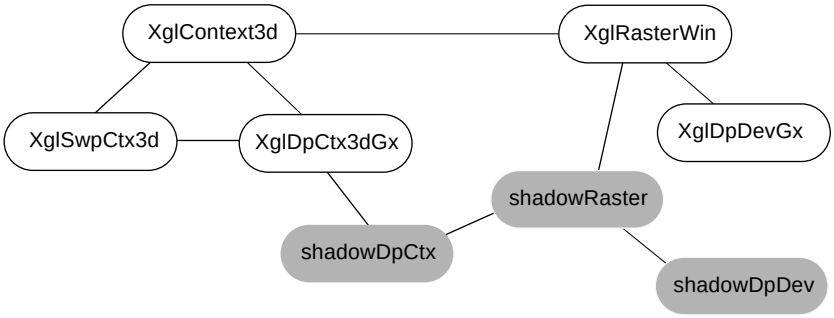


Figure 5-6 Shadow Objects Created for Backing Store

The parent XglDpCtx and the shadow XglDpCtx have the same structure. Each has an `opsVec` pointer, called `currOpsVec`, and two arrays: the `opsVec` array and the `opsVecGen` array. Figure 5-7 illustrates the parent device pipeline-context and the backing store device pipeline-context.

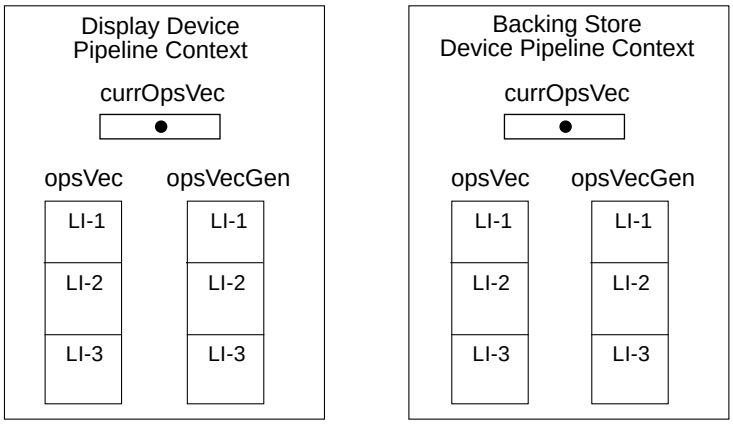


Figure 5-7 Architecture of the Backing Store Device

Creating a Shadow Device

A shadow device is created the first time that backing store is requested. An application sets the `XGL_WIN_RAS_BACKING_STORE` attribute to request backing store. When backing store is requested, the Window Raster calls the device pipeline to inquire if it wants the XGL core to create a backing store device and handle backing store operations. The Window Raster does this with the pipeline interface `dpDev->needRtnDevice()`. If `dpDev->needRtnDevice()` returns `TRUE`, the XGL core will:

1. Get a handle to the backing store memory from the server and create a shadow device, which can either be a Window Raster or a Memory Raster. It then attaches the shadow device to the parent device. The shadow device includes its device-independent and device-dependent parts.
2. Create a shadow device pipeline-context object.
3. Synchronize the raster attributes between the base device and the shadow device.
4. Set the `doRetained` flag in the parent device to `TRUE` to indicate that backing store mode is on and that the server is granting the backing store.

The example code below shows the shadow device being created.

```
void XglRasterWin::setBackingStore(Xgl_boolean
                                   backingstore_mode)
{
 .....
    if (backingstore_mode) {        // user set to TRUE
        if (backingStore) return;   // backing store aleady set
       // no backing store in server mode or w/ double buffering
        // return if window system is not X
       if (buffersAllocated > 1 || XglXServer::isXServerMode()
           || type != XGL_WIN_X)
           return;

       if (((XglDpDevWinRas*)dpDev)->needRtnDevice())
           backingStore = createRtnDevice();
       else {
           backingStore = TRUE;
       }
```

```
    }
    else if {                   // user set to FALSE
         if (shadowDevice) {    // supporting backing store
            ....
             // free backing store device components,
            // reset pointer(s) to NULL
            destroyRtnDevice();
        }
         if (doRetained) {
            drawable->unGrabRetainedWindow();
            doRetained - FALSE;
        }
        backingStore = FALSE;
    }
}
```

Note that there are two possible scenarios for the creation of the shadow interface manager when the Context and Device are associated:

- If the Context and Device have not previously been associated, the pointer to the device pipeline-context object for the Context and Device pair cannot be retrieved from the parent Device. In this case, the parent Device will create the parent device pipeline-context object. If backing store is set, the parent Device will ask the shadow device to create a shadow device pipeline-context for the Context object and then attach it to the parent device pipeline-context.
- If the Context and Device have previously been associated, the pointer to the device pipeline-context object for the Context and Device pair is retrieved from the parent Device.

If a device pipeline can handle backing store itself, it can return FALSE to `dpDev->needRtnDevice()`, in which case all backing store operations become the responsibility of the device pipeline or its underlying framework. For example, because the PEXlib pipeline can rely on the server to handle backing store operations, it returns FALSE to `needRtnDevice()`.

Tracking Backing Store Changes in the Server

Since the server can potentially drop backing store support or switch the allocation of backing store memory to another device, the XGL core must track the changes in backing store on the server side for every `WIN_LOCK()` call.

Whenever a change is detected, the XGL core destroys the existing shadow device and shadow device pipeline-context object for that window (device). New shadow device and shadow device pipeline-context objects are created if new backing store memory is allocated by the server. This operation can be rather time-consuming if it happens in the middle of a rendering call, though it rarely occurs.

Rendering Into the Backing Store Device

When a rendering call occurs and backing store is enabled, XGL first renders into the parent device and then renders into the shadow device. The rendering call is duplicated in order to render into both devices. To render into backing store, the device independent code sets the `currOpsVec` pointer (used by the API wrapper to call a device pipeline function) to point to the `opsVecGen` array instead of the the `opsVec` array. The `currOpsVec` points to the `opsVecGen` only in the parent device. The shadow device's `opsVec` is called directly from the routine pointed to by the `opsVecGen`, and its `currOpsVec` pointer is not used. Figure 5-8 illustrates rendering into backing store.

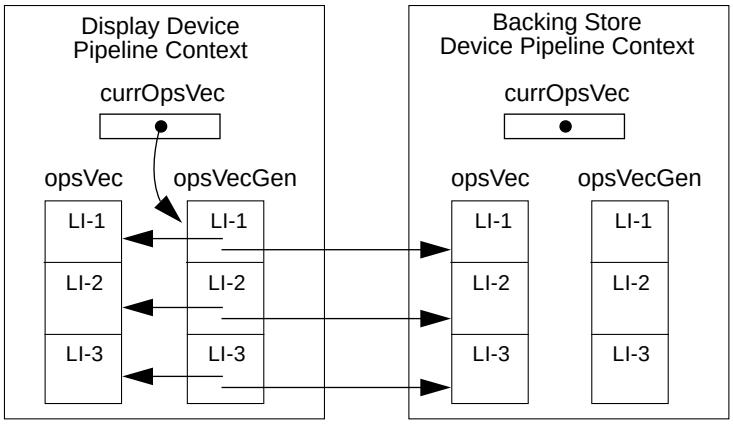


Figure 5-8 Rendering into the Backing Store Device

Since both the parent XglDpCtx and shadow XglDpCtx derive from the base class XglDpCtx, a backing store device pipeline doesn't know it is being used for backing store. In exactly the same way as the parent device, the shadow device will either try to render through hardware, or will call the software pipeline if it doesn't know how to render a given primitive-attribute combination.

As a result, any attribute changes are sent to both the parent device and the shadow device to keep them synchronized. Since the object set function pointer is also in both `opsVec` arrays, when backing store is on, attribute setting is handled through the `opsVecGen` array, which calls the backing store version of the object set function. This function calls object set for both the front and backing store pipelines, keeping them synchronized.

The Context `do_retained` parameter is used only by the device independent code and the software pipeline. By default, `do_retained` is `FALSE;` the Context sets it to `TRUE` when backing store is enabled, and it remains `TRUE` until the processing first gets into parent device's pipeline. Once the rendering call is sent to the parent device, `do_retained` is set to `FALSE`.

If a device pipeline calls the software pipeline, for example at LI-1, the software pipeline processes only once for both the parent device and the shadow device at each LI layer until the parent device successfully renders. That is, if the parent device can only render the primitive at LI-2, then the LI-1 software pipeline processing is done only once for both the parent device and the shadow device.

The code example below shows the `opsVecGen li1MultiPolylineGen()` function rendering into both the parent device and the shadow device if backing store is enabled and rendering only once to the software pipeline if the device pipeline calls the software pipeline.

```
void XglDpCtx3d::li1MultiPolylineGen(Xgl_bbox*    bounding_box,
                                     Xgl_usgn32   num_pt_lists,
                                     Xgl_pt_list* pl)
{
    if (error_checking) {
        do_error_checking()
    }

    if (backing_store_on && !picking) {
        WIN_LOCK(drawable);
    }
```

```
    // Call the parent device to render to
    // the visible part of the screen.
    //
    // NOTE: Add an additional Xgl_boolean (TRUE) parameter in case
    //       the parent device pipeline calls swp and uses the DI
    //       default function.
    //
    (this->*((void(XglDpCtx::*)
             (Xgl_bbox*,Xgl_usgn32,Xgl_pt_list*,Xgl_boolean))
             (opsVec[XGLI_LI1_MULTIPOLYLINE]))
        )(bounding_box,num_pt_lists,pl,TRUE);

    if (backing_store_on && !picking) {
        // If the parent device pipeline handled the call,
        // call opsVec of shadow device dpBs to handle rendering
        // into backing store device
      if (opsVec[XGLI_LI1_MULTIPOLYLINE] != //device didn't call swp
          (void (XglDpCtx::*)()) (&XglDpCtx3d::li1MultiPolyline)) {
            if (dpBs && <other optimization testing>)) {
                     //and shadow device exists
               swp->setDpCtx(dpBs);  // switch dp pointer in swp to
                                     // dp backing store device

                (dpBs->*((void(XglDpCtx::*)
                         (Xgl_bbox*,Xgl_usgn32,Xgl_pt_list*))
                         (dpBs->opsVec[XGLI_LI1_MULTIPOLYLINE])
                         ) //call opsVec entry in shadow dpCtx
                        )(bounding_box,num_pt_lists,pl);

                swp->setDpCtx(this); // reset dp pointer in swp
                                     // to parent dp
            }

            WIN_UNLOCK(drawable);
        }
    }

    if (asap_mode_on)
        // do flush stuff
    }
}
```

The default opsVec li1MultiPolyline() function of the base class XglDpCtx is shown below. The if statement should only be true if the device's doRetained flag is TRUE, and the function is called directly from the corresponding LI opsVecGen function from the parent device as a result of a default call to the software pipeline (and not picking).

In order to distinguish the full LI opsVecGen software pipeline call from the multiple calling of the default device pipeline functions from the software pipeline as a result of partial calls to the software pipeline, an additional Xgl_boolean gen_punt parameter is added to the LI-1 and LI-2 default functions:

```
// Default function pointed to by opsVec array
// if device pipeline wants to call the software pipeline.
void XglDpCtx3d::li1MultiPolyline(Xgl_bbox*    bounding_box,
                                  Xgl_usgn32   num_pt_lists,
                                  Xgl_pt_list* pl,
                                  Xgl_boolean  gen_punt)
{
    // The doRetained field corresponds to XGL API attribute for
    // XGL_WIN_RAS_BACKING_STORE && the server granting backing
    // store.
    // Also, no picking for backing store case.
    // gen_punt has to be tested *after* doRetained

    if (device->getDoRetained() && gen_punt
                               && !ctx->getPickEnable()) {
        // release the lock in li1MultiPolylineGen
        WIN_UNLOCK(drawable);

       swp->li1MultiPolyline(bounding_box, num_pt_lists, pl, TRUE);
    }
    else
        swp->li1MultiPolyline(bounding_box, num_pt_lists, pl);
}
```

There is no need to add this extra parameter to LI-3 because calling the software pipeline is not allowed in LI-3, and an error will be issued in that case. As noted on page page 82, the opsVecGen array is static; in other words, a device pipeline cannot change function pointer entries in this array.

The Xgl_boolean argument is FALSE by default, so any calls directly from one XglDpCtx function to another XglDpCtx function do not need to pass the extra parameter.

```
void  XglDpCtx3d{device}::li2TriangleList (XglPrimData* pd)
{
        // ...
        // break Tlist into Msp

       // call base default function with default value FALSE if not
       // overridden, else call the derived class function that does
        // not have the extra parameter
        li2MultiSimplePolygon(pd);
        //....
}
```

However, if the device pipeline calls these functions through the `opsVec` array, it must add an extra `Xgl_boolean` parameter to the casting and pass in `FALSE` as the argument value in case the default device pipeline function is called. Even if the derived class has overridden the function (which takes one less argument), the extra argument passed in will simply be ignored.

```
void  XglDpCtx3d{device}::li2TriangleList (XglPrimData* pd)
{
        // break Tlist into Msp

        // Call Msp
        (this->*((void(XglDpCtx3d::*)(XglPrimData*, Xgl_boolean))
                 (this->opsVec[XGLI_LI2_MULTI_SIMPLE_POLYGON])
                ))(pd, FALSE);
        ....
}
```

Propagation of API Changes to the Backing Store Device

The parent device is responsible for propagating any pertinent API changes to the shadow device. These operations are done in the XGL core without the device pipeline's intervention, unless set functions are explicitly called. The parent Window Raster device will reflect all relevant changes to the shadow device if the shadow device exists. For example, a window resize event could trigger a change to the memory allocation by the server. The XGL core will handle the corresponding change to the shadow XGL device by destroying the old XGL shadow device and creating another one if needed.

These operations involve explicit create and set calls to the device pipeline. However, the device pipeline does not need to be aware of the internal backing store operations.

Backing Store Support for the Z-Buffer and Accumulation Buffer

For devices that use a software Z-buffer or accumulation buffer, the server only provides backing store during an expose event for the image buffer, not for the Z-buffer or accumulation buffer. XGL enables the sharing of software Z-buffers and/or accumulation buffers with the backing store device. This allows the software Z-buffer and accumulation buffer to stay synchronized with the backing store device.

Devices that may be used as backing store devices (for example, a Memory Raster) and that use software Z-buffer and/or accumulation buffer must implement `setSwZBuffer()` and `setSwAccumBuffer()` so that the XGL core can call these functions and pass in the parent's software buffer addresses during backing store device creation.

For devices that use software Z-buffer and accumulation buffer, setting HLHSR mode on or calling the accumulation function for the first time may trigger the creation of the software buffer in the parent device. The XGL core is responsible for passing the addresses of the buffer memory from the parent device to the backing store device for sharing.

The backing store device should cache the parent device pointer that is passed in during backing store device creation time. Before creating any Z-buffer or accumulation buffer, these device pipelines should ask the parent device for any software buffer address through `parent_dev->getSwZBuffer()` or `parent_dev->getSwAccumBuffer()` to make sure it is sharing those buffers as much as possible.

Note that devices with hardware Z-buffer or accumulation buffer cannot share these buffers with backing store. Thus, these devices may not want to enable backing store when HLHSR mode is on or when accumulation is needed. See the `XGL_WIN_RAS_BACKING_STORE` reference page in the *XGL Reference Manual* for more information.

Error Handling 6

This chapter describes the XGL error handling mechanism.

Design Goals

Error processing in XGL was designed to do the following:

- Allow an application to provide its own error reporting function(s)
- Allow a device pipeline to provide its own error file containing error messages specific to the pipeline
- Provide for internationalization of error message strings
- Maintain state information when an error is detected that defines the error, its cause, and where it occurred

Overview of Error Handling

Error handling in XGL can be viewed in two ways: externally and internally. The external view describes what the user sees at the API level when an error occurs. The internal view describes how the error is handled by functions that are not visible at the API level.

6

External Error Handling Mechanism

Externally, from the API point of view, when an XGL application causes an error, the error notification function determines the application-visible response. Because this function is settable by the application, its response can vary depending on its current definition. The default error notification function sends an error message to `stderr`. For example, the following message is produced by the default function for a `malloc` error that occurs within an `xgl_polygon()` call from a 3D Context:

```
Error number di-1: malloc or new failed: out of memory
Operator: xgl_polygon
Object: XGL_3D_CTX
```

Error Notification Function

The XGL-provided default error notification function is defined in `Error.h`. When an error occurs, this function sends an ASCII string to the standard error output. The string consists of the XGL error code associated with the error, a message describing the error, the XGL operator being executed when the error was detected, the XGL object being used when the error was detected, and other optional information.

XGL enables an application to supply its own error notification function for filtering errors and reporting them to the application program. The following code fragment shows an example of an application error notification function.

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

static Xgl_sgn32 newNotify(system_state)
    Xgl_sys_state system_state;
{
    Xgl_error_info      info;
    int               n;

    xgl_object_get (system_state, XGL_SYS_ST_ERROR_INFO, &info);

    printf (“id      = %s\n”, info.id);
    printf (“msg     = %s\n”, info.msg);
    printf (“cur_op  = %s\n”, info.cur_op);
    printf (“cur_obj = %s\n”, info.cur_obj);

    n = atoi (info.id);
    if (n < 100) {
        printf (“Non-Recoverable Error!\n”);
    } else {
        printf (“Recoverable Error!\n”);
    }
    return(1);
}
```

This function uses the information in the *Xgl_error_info* structure returned by XGL_SYS_ST_ERROR_INFO to handle errors. Any of the eight fields in the structure can be used to filter errors. The System State attribute XGL_SYS_ST_ERROR_NOTIFICATION_FUNCTION sets the application-specific error function. The interface for the error notification function is as follows:

```
my_error_notify_func(Xgl_sys_state   sys_state);
```

Error Types and Categories

XGL errors are grouped into the five categories listed in Table 6-1 on page 108. These error categories enable an application programmer creating a new error notification function to detect a particular error group and respond accordingly. Errors in these categories can be further classified into

RECOVERABLE or NON-RECOVERABLE error *types*, as described in Table 6-2. This table also describes the relationship between error types and categories.

Note that the XGL default error notification function does not use the error types and categories. These are provided for developers who want to trap specific error types or categories in their own error notification function.

Table 6-1 Error Categories

Category	Description
SYSTEM	Internal errors, unsupported features, and errors that generally cannot be fixed by changing the application.
CONFIGURATION	Errors caused by improper installation or configuration of XGL (such as a `.so` file not found).
RESOURCE	Unavailable resource errors including both hardware and software resources (such as memory, shared memory, window ID, frame buffer).
ARITHMETIC	Arithmetic exceptions (such as an error resulting from dividing by 0 or taking the square root of -1).
USER	Errors caused by invalid function parameters, non-existent user files, or situations that may be caused by application program logic errors.

Table 6-2 Error Types

Type	Description
NON-RECOVERABLE	XGL immediately aborts processing and returns control to the caller. Includes SYSTEM and CONFIGURATION errors, most RESOURCE errors, and some ARITHMETIC errors.
RECOVERABLE	XGL makes assumptions about what the application intended to do. Includes some RESOURCE errors, most ARITHMETIC errors, all USER errors.

Error Message Files

Binary-encoded files containing the English versions of error message strings are located in:

{*path*}/{*LANG*}/LC_MESSAGES/file.mo

where {*path*} is $XGLHOME/lib/locale if $XGLHOME is set, or /opt/SUNWits/Graphics-sw/xgl-3.0/lib/locale if $XGLHOME is not set. {*LANG*} is en for English language messages. Error message files that have been localized to the native language of the user will be found in other {*LANG*} directories.

More than one error message file may exist in this directory. At the very least, there is a file called xgl.mo, containing device-independent error messages. There may also be a file named xgl<*company abbrev*><*pipeline abbrev*>.mo, which contains error messages specific to the device pipeline. The directory path of the error message file is specified internally and cannot be set or retrieved via the XGL API.

Internal Error Handling Mechanism

Internally, when XGL detects an error, it calls an internal error handling function. This function assigns values to error attributes, searches for the error file that contains localized error messages, and retrieves the appropriate error message string. When the error message string is retrieved, the error handling function calls the application-settable error notification function for further processing of the error.

Error processing is handled centrally in a device-independent manner by the System State object. For maintainability, however, most error-specific attributes and methods are defined in a separate Error class.

The System State class defines the API interface functions used for error processing and contains error attributes exposed at the API. The Error class contains the default error notification function, functions that initialize the path to the error file, and a function used for error notification when System State

creation fails. The Error class also defines the error attributes for the System State object. Other attributes in the Error class define state information that is saved when an error occurs. These are shown in Table 6-3.

Table 6-3 State Information Saved in an Error Object

Information	Description
Type	RECOVERABLE or NON-RECOVERABLE
Category	SYSTEM, CONFIGURATION, RESOURCE, ARITHMETIC, USER
ID	An error number in the form: *<pipeline abbrev.>-##*
Message	An error message string
Operator	XGL API operator in use when the error occurred
Object	XGL object in use when the error occurred
Operands	Two operands of error notification

When the System State object is initialized at `xgl_open()`, it instantiates an Error object and saves a pointer to the object. The Error object instantiation also defines the default error notification function.

When an error is detected, one of two internal error handling functions, `reportError()` or `reportDiError()` in the System State object, is called. These functions store current state information into the Error object. The error handling function invokes another internal routine, `errorHandler()` in the Error object, which gets the state information from the Error object and passes it to the current error notification function. It then invokes the error notification function to complete the error processing.

The state information in the Error object is overwritten when the next error occurs. Thus, the information stored in the Error object and referenced by the System State object is valid only during the invocation and execution of the error notification function.

Note – The Error object must be accessed immediately after the occurrence of an error, or the state information used by the error notification function may not be accurate.

XGL Coding Guidelines 7

This chapter discusses XGL naming conventions for C++ constructs and internal C++ classes, and presents the coding conventions for XGL member data accessor functions.

Naming Conventions for C++ Constructs

The intent of the XGL naming scheme is to provide consistency, simplicity, and ease of distinction between C and C++ features. In this scheme, XGL identifies C++-specific constructs by concatenating multiple-word names with the first letter of each word capitalized, while other constructs that fall within the C domain preserve the C underscore naming style to ease migration and maintain C API compatibility. Identifiers in the global name space are prefixed with the package name to avoid conflicts.

The specific naming conventions are as follows:

- Class names: `XglFooClass` – Class names begin with the `Xgl` prefix to avoid name conflicts. The first letter of a class name is capitalized, and if the class name is a compound word, the first letter of each subsequent word is capitalized.
- Class member data: `dataMember` – The first letter of a member data name is lower case, but if the member data name is a compound word, the first letter of each subsequent word is capitalized. Capitalization distinguishes member data and local variables/function arguments within a member function when the variable names are compound words. No further effort is made to distinguish member data from local variables or function

arguments, however, as the latter declarations are always within the same scope and should be easy to locate. Since global variables (which should be avoided) always start with a capital letter, no confusion will occur.

- Member functions: `memberFunc()` – The first letter of a member function name is lower case, but if the member function name is a compound word, the first letter of each subsequent word is capitalized. Capitalization distinguishes member functions from non-member C++ functions and maintains consistency in naming C++-specific constructs.
- Regular (global or file static) C++ functions (non-extern "C"): `XgliGlobalFunc()` – The first letter of regular C++ function is capitalized, and the first letter of each subsequent word in a compound function name is also capitalized.
- Macro values, enumerated fields, and constants: `XGL_CONST_VALUE` – Macros, enumerated fields, and constants are upper case with underscores separating the words. This follows the C convention for `#define` constants; however, `#define` should not be used to define constants except in the C API. Instead, constants and enumerated values should be used to allow for proper type-checking and scope-related constants.
- Macros and global inline functions: `XGL_INLINE_FUNC()` – Inline function names are upper case. Inline functions should be used in place of macro functions except in the C API. Inline functions are expanded at compile time and produce semantically correct results as well as allowing type checking.
- Global variables: `Xgli_global_var` – In general, all global variables should be scoped within a class as static member data and accessed only through static member functions. However, if it is determined that a global variable is necessary, then the first letter of the global variable name is capitalized. Compound words are constructed with underscores separating the words. The `Xgl` package prefix is added where appropriate.
- Local variables and function arguments: `local_or_param` – Local variables and function arguments are lower case, with multiple-word variables separated by underscores. The use of underscores distinguishes local variables and function arguments from member data inside member functions.
- Typedef, enumerated types, and C API structures and unions: `Xgl_typedef` – Although `typedef` and `enum` identifiers are also type names, and structures and unions are semantically similar constructs to

classes, `typedef`s and `enum`s are so extensively used in the existing C API that their naming should follow the C convention, which is the first letter capitalized and compound words separated by underscores. For the same reason, this rule extends to structure and union fields in the API. It is useful to be able to distinguish "pure" C structures and unions that do not use any C++ features in this way, although C++ does not differentiate them semantically. For C++-specific usage, `class` should always be used in place of `struct`, while the `union` type should be avoided as it defeats strong type checking.

- Extern "C" functions: `xgl_object_create()` – To preserve the C API interface, all `extern` "C" functions used as C wrappers should use the C function naming style, which is lower case with compound words separated by underscores. This also allows clear distinction from their C++ counterparts.

- Pointers and reference variables: `int* a; int& b = *a`
  For pointers and reference variables, bind * and & to the type except in lists. The placement of * and & should be consistent. It is visually less confusing to declare a C++ reference type as `int& b` than as `int &b` (since `&b` is like taking the address in C). It is more coherent to use

  ```
  A& XgliGlobalFunc (A& a, B&);
  ```

  than

  ```
  A & XgliGlobalFUnc (A &a, B &);
  ```

  Therefore, both * and & should be bound to the type.

  One exception to this rule is when there is a list of declarators, as in:

  ```
  A *x, &y=*x, *z;
  ```

  in which * and & should be bound to the variables. Note that use of a declarator list is strongly discouraged. The recommended style is to declare one variable at a time with a comment for each to the right of the variable.

Summary of Naming Conventions for C++ Constructs

Table 7-1 provides a summary of the naming conventions.

Table 7-1 Summary of Naming Conventions for C++ Constructs

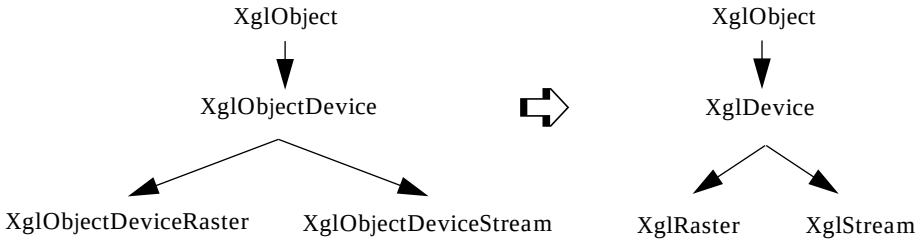
Construct	Convention	Example
Class names	First letter of each word capitalized with remainder of word lowercase; Xgl prefix	`XglFooClass`
Class member data	First letter lowercase with following words mixed case	`memberData`
Member functions	First letter lowercase with following words mixed case	`memberFunc()`
Regular (global or file static C++ funtions (non-extern “C”)	First letter capitalized with following words mixed case	`XgliGlobalFunc()`
Macro values, enumerated types, constants	All capital letters with underscores between words	`XGL_CONST_VALUE`
Macros and global inline functions	All capital letters with underscores between words	`XGL_INLINE_FUNC()`
Global variables	First letter capitalized (Xgl prefix where appropriate) and underscores between words	`Xgli_global_var`
Local variables and function arguments	All lower case with underscores between words	`local_variable`
Typedefs, enumerated types, and C API structures and unions	First letter capitalized with underscores between words	`Xgl_typedef`
Extern “C” functions (for example API wrappers)	All lowercase with underscores between words	`xgl_object_create()`

Pointers and reference variables	Bind pointers and reference types to type except in list	`int* a;`
		`int& b = *a;`
		`A& XgliGlobalFunc (A& a);`
		exception:
		`A *x, &y = *x, *z;`

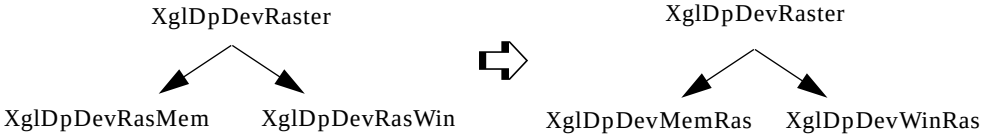
Naming Conventions for C++ Internal Classes

The XGL C++ class naming conventions are as follows:

1. All class names are prefixed with `Xgl` to avoid name collisions. For example, use XglDevice rather than Device.
2. Compound class names are combined without underscores and with the first letter of each word capitalized.
3. Classes are named in top-down order in the inheritance hierarchy. For example, the 2D subclass of XglContext is named XglContext2d.
4. Common prefixes are removed when the meaning is obvious, as in the following example:



5. Abbreviations are used where appropriate. For example, Context can be abbreviated as Ctx, Device can be abbreviated as Dev, and XglDevicepipelineDevice as XglDpDev.
6. Follow common naming order, as in the example below:



7. Prefix pipeline-related classes with Swp for software pipeline classes and Dp for Device pipeline classes, as in XglDpDev, or XglDpMgr.
8. Follow the mixed-case rule even for acronyms. For example, use XglCgm rather than XglCGM.

7

Coding Conventions for set() and get() Interfaces

The following conventions have been adopted by XGL for the C++ member data accessor functions `set()` and `get()` both internally and externally (C++ API level).

Conventions for set() Member Functions

Following are the conventions for the `set()` member functions.

1. Pass by value all base types and typedefs of base types.

   This convention includes all signed and unsigned integer base types, character base types, as well as float, double, etc. For example,

   ```
   void xgl_object_set(Xgl_ctx, XGL_CTX_CLIP_PLANES,
              Xgl_clip_planes, 0);
   ```

   maps to C++:

   ```
   void XglContext::setClipPlanes(Xgl_clip_planes);
   ```

2. Pass "pure" C structures or unions as C++ object references.

   For example,

   ```
   void xgl_object_set(Xgl_ctx, XGL_CTX_BACKGROUND_COLOR,
            Xgl_color*, 0);
   ```

   maps to C++:

   ```
   void XglContext::setBackgroundColor(const Xgl_color& color){
          backgroundColor = color; }
   ```

   This is preferable to manually dereferencing a pointer every time you need to reference it as a structure, as in:

   ```
   void XglContext::setBackgroundColor(const Xgl_color* pcolor) {
            backgroundColor = *pcolor;  }
   ```

3. Pass XGL objects, function pointers, and character strings as pointers.

   For example,

   ```
   void xgl_object_set(Xgl_ctx, XGL_CTX_DEVICE, Xgl_dev, 0);
   ```

   maps to C++:

   ```
   void XglContext3d::setDevice(XglDevice*);
   ```

Note that `Xgl_dev` is a pointer itself.

4. Pass array types as arrays.

   For example:

   ```
   void xgl_object_set(Xgl_3d_ctx, XGL_3D_CTX_LIGHTS,
          Xgl_light[], 0);
   ```

   maps to C++:

   ```
   void XglContext3d::setLights(const XglLight[]);
   ```

Conventions for get() Member Functions

Following are the conventions for the `get()` member functions.

1. Return by value for base types or typedefs of base types.

   For example,

   ```
   void xgl_object_get(Xgl_ctx, XGL_CTX_CLIP_PLANES,
           Xgl_clip_planes*);
   ```

   maps to C++:

   ```
   Xgl_clip_planes XglContext::getClipPlanes() const;
   ```

   The `const` means that the XglContext object does not change after calling its `getClipPlanes()` member functions. `const` should be used wherever appropriate; otherwise, member functions cannot be called by class objects that are declared as `const`.

2. Return “pure” C structures or unions as object references.

   For example:

   ```
   void xgl_object_get(Xgl_ctx, XGL_CTX_BACKGROUND_COLOR,
           Xgl_color*);
   ```

   maps to C++:

   ```
   const Xgl_color& XglContext::getBackgroundColor() const;
   ```

   `const` is used for return types if the internal structures are not supposed to be modified, and the user has to make a copy. So in a C++ program, you can have:

   ```
   Xgl_color mycolor = ctx->getBackgroundColor();
   ```

Copying from the internal background color in the Context object to `mycolor` happens during the assignment.

3. Return XGL objects, function pointers and character strings as pointers.

   For example:

   ```
   void xgl_object_get(Xgl_ctx, XGL_CTX_DEVICE, Xgl_dev*);
   ```

   maps to C++:

   ```
   XglDevice* XglContext::getDevice() const;
   ```

4. Follow the XGL API style for data that involve arrays of objects.

   This is an exception to the convention of returning results through function return types, as there is no implicit way to copy an array. More importantly, the only way to ensure that the internal array objects (pointers to lights in the example below) cannot be modified is to provide a copy of the whole array (of pointers), since return types like `const Xgl_light[]` are not allowed. Hence, array types or structures that contain pointers to array types should continue to be passed and retrieved as function parameters as in the XGL API.

   For example:

   ```
   void xgl_object_get(Xgl_3d_ctx, XGL_3D_CTX_LIGHTS, Xgl_light[]);
   ```

   maps to C++:

   ```
   void XglContext3d::getLights(Xgl_light[]);
   ```

   Note that an exception exists even in XGL 2.0, where an application has to get the value for `XGL_3D_CTX_LIGHT_NUM` before it calls the above function.

   ```
   void xgl_object_get(Xgl_cmap, XGL_CMAP_COLOR_TABLE,
           Xgl_color_list*, 0);
   ```

   maps to C++:

   ```
   void XglCmap::getColorTable (Xgl_color_list&) const;
   ```

Index

X