The Solaris XGL Graphics Library

Technical White Paper

SunSoft
A Sun Microsystems Company

Please
Recycle

Contents

The Solaris XGL Graphics Library 1

Introduction

The Solaris XGL Graphics Library is a mature graphics interface, providing foundation-level geometry graphics support for Solaris-based applications. The XGL Graphics Library is both an Application Program Interface (API) and a geometry foundation library and is just one part of the powerful graphics environment provided as standard with the Solaris 2.x operating system. The latest release of the XGL Graphics Library represents major enhancements to XGL, such as NURBs, texture mapping, transparency and a loadable pipeline architecture.

The *Solaris VISUAL* environment consists of a set of Foundation Graphics Libraries: (XIL and XGL together with PostScript and X) and X extension libraries: (PEXlib, XIElib, DPSlib, and Xlib). This environment provides the most flexible and feature-rich graphics platform in the industry. XIL contains a set of key functions from the fields of image processing and digital video, which address applications and APIs for the imaging, video and related markets. PostScript provides de facto standard PostScript Level 2 rendering within OpenWindows, and X is the industry standard 2-D protocol common to all X Windows environments. These four foundation libraries meet the key requirements of graphics-oriented applications and layered third-party APIs such as PHIGS, GKS, HOOPS, GL, OpenGL, PIKS and many others.

Solaris 2.x is based upon the UNIX System V Release 4 (SVR4) operating system. A key feature of SVR4 is its support for dynamic linking and loading. This capability enables low-level foundation interfaces such as XGL and XIL to dynamically load device handlers associated with underlying display hardware such as framebuffers, video boards and geometry accelerators. The flexibility to dynamic load devices is leveraged through the XGL and XIL libraries' support for a stable device porting interface which greatly simplifies the development and delivery of third-party display hardware on Solaris-based systems. Figure 1.1 illustrates the Solaris VISUAL graphics interfaces.

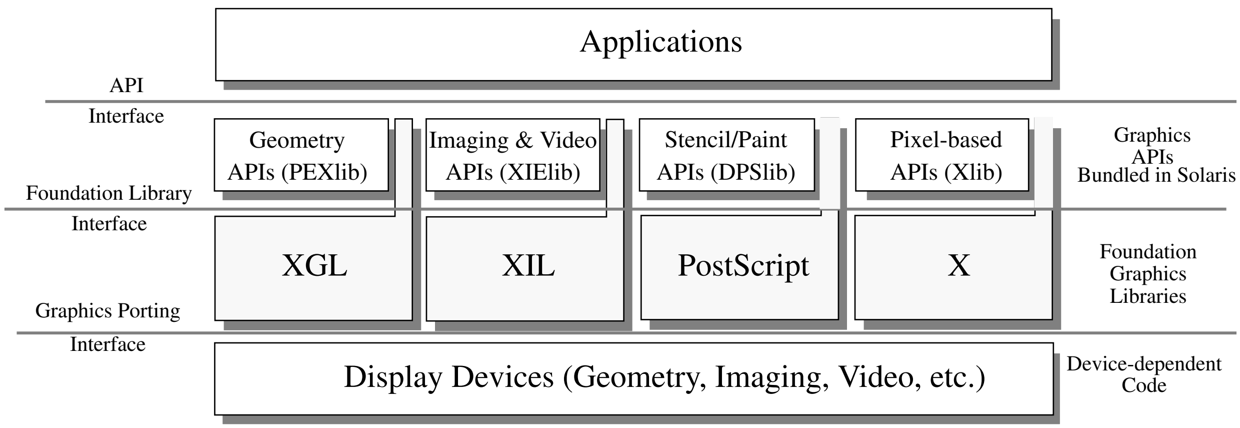


Figure 1.1 The Solaris VISUAL Graphics Environment

By providing foundation-level interfaces, SunSoft focuses on the core requirements of today's sophisticated client-server applications. Additionally, the foundation-level flexibility and functionality enables developers and users to select the best combination of applications, APIs and hardware to fit their needs.

XGL was developed as a foundation library from its inception. Using the most widely accepted standard API functionality as a basis, XGL was developed to address requirements for graphics rendering of applications such as electrical and mechanical design and drafting, mapping, animation, and simulation. The XGL Library provides for optimization of 2-D and 3-D rendering, including high-quality lighting and shading, advanced primitives (such as NURBs and meshes, texture mapping and transparency) and a flexible geometry pipeline.

The XGL Library has an object-oriented interface that hides the low-level details of implementation from the application developer. As such, it is architected such that it may evolve as an interface, permitting new algorithms to be implemented within XGL without incompatibly changing the application program interface bindings. Figure 1.2 illustrates how XGL works with third-party APIs.

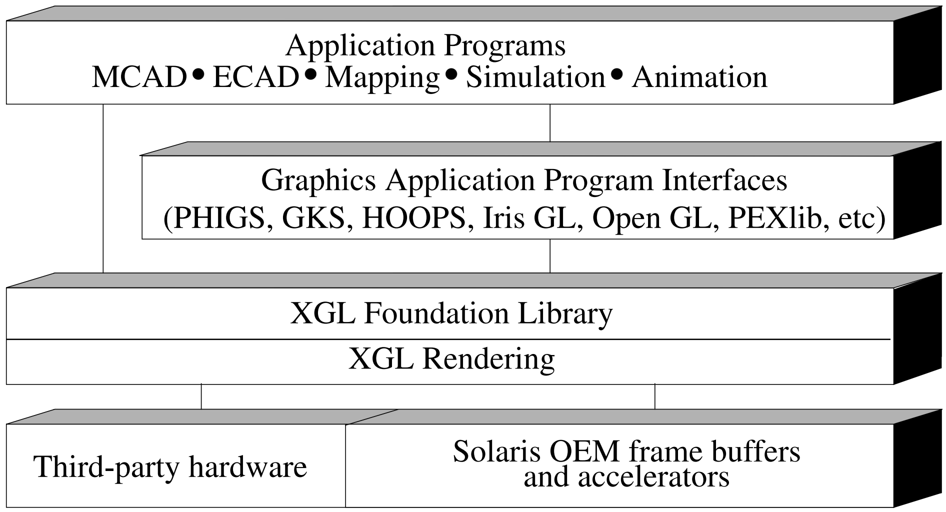


Figure 1.2 Solaris VISUAL Supports the Full Breadth of Open and Third-Party APIs

XGL Architecture Overview

The XGL Library contains two primary components: a software programming interface specification for API and application developers and a standard graphics device porting interface specification, which enables third-party hardware developers to readily support XGL applications.

The XGL Library delivers high functionality and performance to applications and graphics APIs that wish to maintain their own display lists or data structures. It provides:

- Immediate-mode rendering
- Complementary, but separate 2-D and 3-D graphics pipelines
- Object-oriented interface characteristics
- Broad primitive and coordinate type support

The XGL graphics interface is specifically designed to emphasize immediate-mode rendering. Immediate-mode refers to immediate rendering of graphics primitives without the requirement for placing the data in a display list, and using predefined subroutines to manipulate, edit, or search the data set. Using this approach, XGL lets the application organize the data in an efficient manner.

The immediate-mode approach is especially well suited for applications requiring highly interactive characteristics, or maintaining specialized data, including applications in mapping, animation, and simulation.

XGL Core Library and Rendering

At the most basic level, XGL has two components: the XGL core and the XGL rendering pipelines. The core functions as the interface between the applications program and the rendering pipelines. The rendering pipelines turn a geometric data description of an object and its state attributes into pixel data that is displayed on a graphics hardware device or written into memory. Figure 1.3 illustrates these basic components.

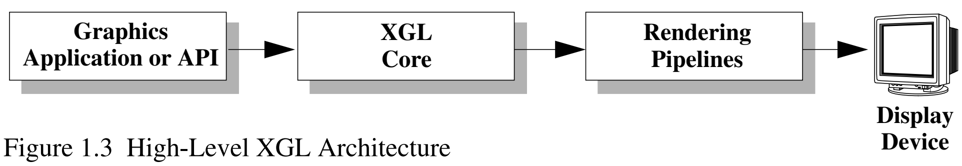


Figure 1.3 High-Level XGL Architecture

XGL Core Library

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

- Creation and management of API graphics objects
- Processing of state changes for its pipelines
- Mapping XGL C API function calls to internal C++ functions

XGL Rendering

The rendering pipelines include one or more device pipelines and a software pipeline. The pipelines are divided into three layers, each responsible for interfacing at specific pipeline locations. The top layer, Loadable Interface 1 (LI-1) specifies a device-independent interface that lies directly below the XGL API. Functions in this layer implement the geometry pipeline for each drawing primitive; they take the points defining the primitive and transform, light (in the 3-D case), and clip the geometry in preparation for the rendering operations in the next layer. The second layer, Loadable Interface 2 (LI-2) is responsible for scan converting more complex primitives like polygons and polylines. The lowest layer, Loadable Interface 3 (LI-3) is responsible for rendering pixels, individually or in spans, on the device.

The default pipeline is the software pipeline which XGL provides for devices that have only minimal acceleration capabilities. The software pipeline consists of functions that implement the geometry pipeline (LI-1), and scan conversion (LI-2) functions in software. A device pipeline can choose to override the default software implementation for any primitive, choosing instead to accelerate the primitive's rendering through the accelerator's own facilities. XGL loadable pipelines are an example of foundation library device handlers - dynamically loaded. Figure 1.4 provides a high-level view of the XGL architecture.

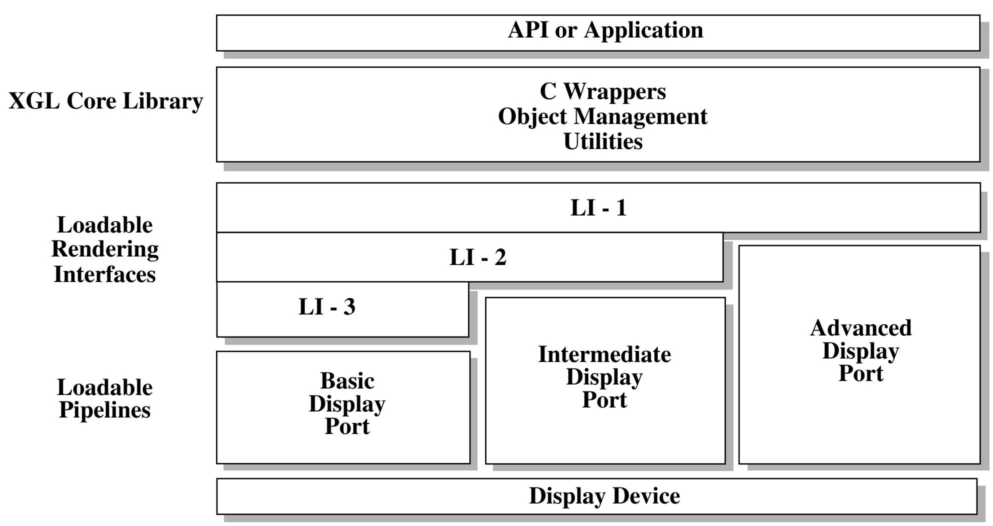


Figure 1.4 XGL Architecture Detail

XGL Enhancements: Differences from Previous Releases

The latest release of the XGL Graphics Library includes the broadest set of enhancements made to the library to date. These enhancements include:

- Significant feature enhancements including new primitives such as triangle lists and NURBS surfaces, and additional features including texture mapping, transparency, antialiasing, accumulation buffering, and backing store
- Re-design of the library structure to support independent loadable pipelines (device handlers) for portability and ease of device integration
- Enhancements to geometry caching, coordinate type support, double buffering and object management
- Performance enhancements throughout
- Migration to the Solaris 2.x distributed client-server operating system
- Support on both SPARC and x86 microprocessor architectures
- Complete re-implementation of the library internals in C++

Major architectural changes in the XGL Graphics Library have been made to address the demands of providing flexible, open and high performance geometry rendering capabilities for applications under Solaris 2.x. Support for loadable pipelines extends XGL to enable easier implementation, support and evolution of graphics display hardware.

Further enhancements to XGL's Core Library and Rendering facilities are designed to extend the interface to address the broader, demanding requirements of sophisticated 3-D applications as well as providing the desired base level features desired by large numbers of 2-D and 3-D application developers.

The XGL Architecture

XGL Architecture Design Goals

The latest release of XGL greatly emphasizes portability and stability. The principal goal of this release was to facilitate porting of XGL to the widest possible range of graphics applications and hardware platforms. In order to encourage the porting of XGL, and the porting of applications and APIs to XGL, this release defined an interface which will be binary compatible with future releases for many years. The XGL architecture was designed to fulfill the following goals:

- Base XGL API semantics on existing standards where possible, to simplify application development and ensure efficient mapping between third-party APIs and XGL
- Employ an object-oriented programming model for ease of future extensions to XGL including new features and capabilities, and broader device support
- Define internal interfaces that enabled structured porting of XGL to new display devices (frame buffers, accelerators, etc.)
- Clearly define the relationships between internal objects
- Optimize XGL Core performance to ensure that graphics accelerators are able to achieve their maximum possible performance

Loadable Pipelines

The XGL loadable pipelines reside in modules that are dynamically loaded at runtime. When an XGL application begins execution, XGL determines which device the application is running on and loads the pipeline library needed to control that device. Hence device support is not contained within XGL, but is loadable at runtime.

XGL provides a device pipeline structure with considerable flexibility in how it renders a primitive. The pipeline can be written to:

- Fully accelerate a primitive from model coordinates to device coordinates
- Partially accelerate a primitive and fall back on the software pipeline for some rendering tasks
- Not accelerate the primitive and allow the software pipeline to perform all rendering tasks

Any time a primitive or rendering attribute for a primitive is not supported by underlying hardware, the device pipeline can fall back on 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.

The pipeline interface layers permit hardware-specific functions to be substituted for XGL-provided code at differing levels of complexity. This allows pipeline writers to take advantage of specific capabilities and performance of their hardware device.

Writing device pipeline functions at the LI-1 layer is complex and offers the greatest amount of freedom, since functions at this layer must handle all aspects of processing an XGL primitive and all rendering operations, including scan conversion and pixel painting. Functions at the LI-2 layer assume responsibility for rendering but leave geometry processing operations to the XGL software version of the LI-1 layer. Functions at the LI-3 layer require implementing of 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. Figure 2.1 shows the stages of the XGL device and software pipelines, and Figure 2.2 illustrates how the individual LI layers work.

A minimal port of XGL to a device must include functions at the LI-3 layer, since functions at this level are not provided by the software pipeline.

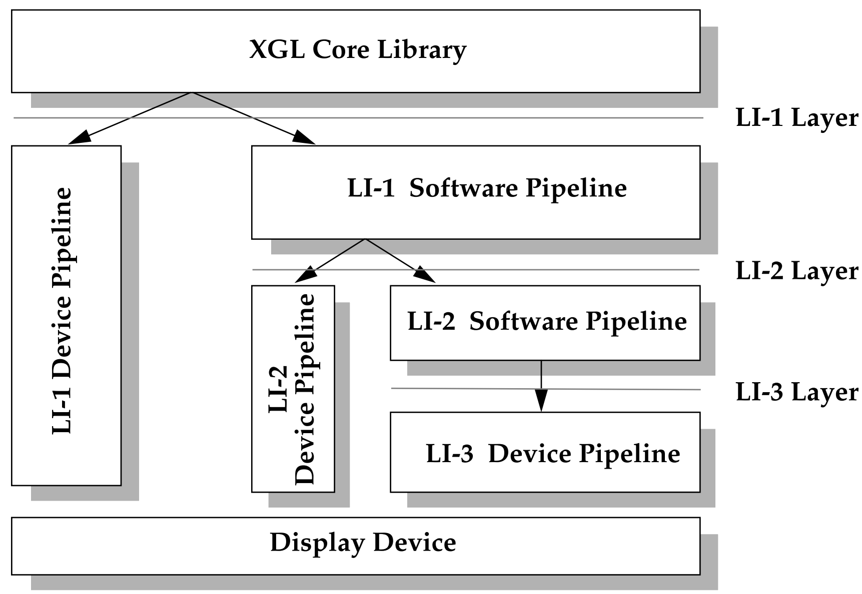
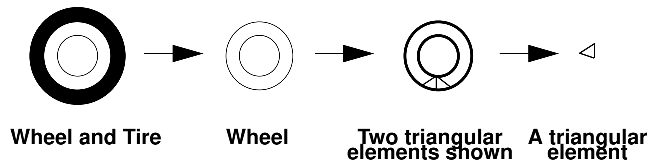
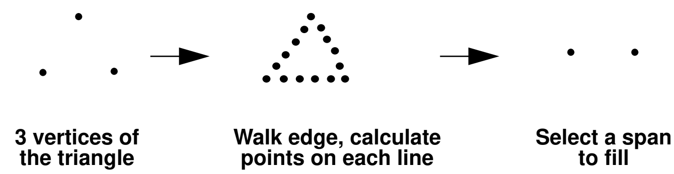


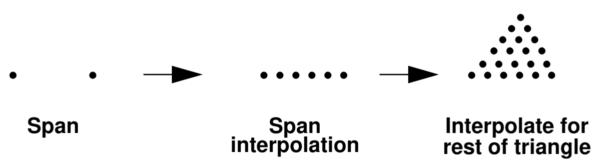
Figure 2.1 XGL Pipeline Architecture



Geometrical objects are generally approximated by quadrilaterals or triangles. Sometimes these simple primitives are chained together as a mesh or strip, which the graphics accelerator may recognize and render efficiently. Managing chained primitives, as well as calculating transformations, lighting and shading are done at the LI-1 layer.



The fundamental representation of the geometrical primitive includes its vertices, the x,y,z coordinates of its endpoints. Some graphics accelerators can rapidly compute the intermediate pixels along the edges of the primitive. Calculations for edge walking and span interpolation are performed at the LI-2 layer.



The most basic task is the computation of intermediate pixels within the primitive. All graphics accelerators can interpolate pixel values along each span. A span is essentially a portion of a line (or row) of the display, which the primitive overlaps. The rendering of individual spans is done at the LI-3 layer.

Figure 2.2 XGL Functions and Device Porting Layers

Device Pipeline Architecture

The XGL Graphics Library does not possess 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. A set of well defined, abstract interfaces are used to link the device pipeline with the XGL core.

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. The loadable pipeline design employed by XGL was developed to explicitly address the wide range of geometry-oriented graphics devices used today, and to provide a foundation for future display technology.

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 API-level device-independent code to the device pipeline code. As part of the process of writing a device pipeline, the writer must subclass the XGL-provided abstract interfaces and implement them to provide device-specific classes and objects.

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, acting as a clearing house for attribute information and doing 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 wanted to render both 2-D and 3-D geometry.

XGL provides a set of objects which manage the pipelines and rendering. These objects include:

- Device Pipeline Context Objects (DpCtx): Objects connecting device-independent Context and Device objects with device-specific pipeline code, and containing rendering functions for the specific hardware
- Device Pipeline Device Objects (DpDev): Objects which manage the DpCtx objects corresponding with each Device object, and which maintain device-dependent information for the pipeline, providing a communication link to the XGL core

- Device Pipeline Manager Objects (DpMgr): Objects which handle various categories of devices, such as multiple frame buffers
- Device Pipeline Library Objects (DpLib): Objects representing the entire device pipeline and which enable multiple DpMgr objects to share hardware and software resources

Figure 2.3 illustrates the relationship among XGL high-level pipeline objects for a multiple frame buffer, multiple application scenario.

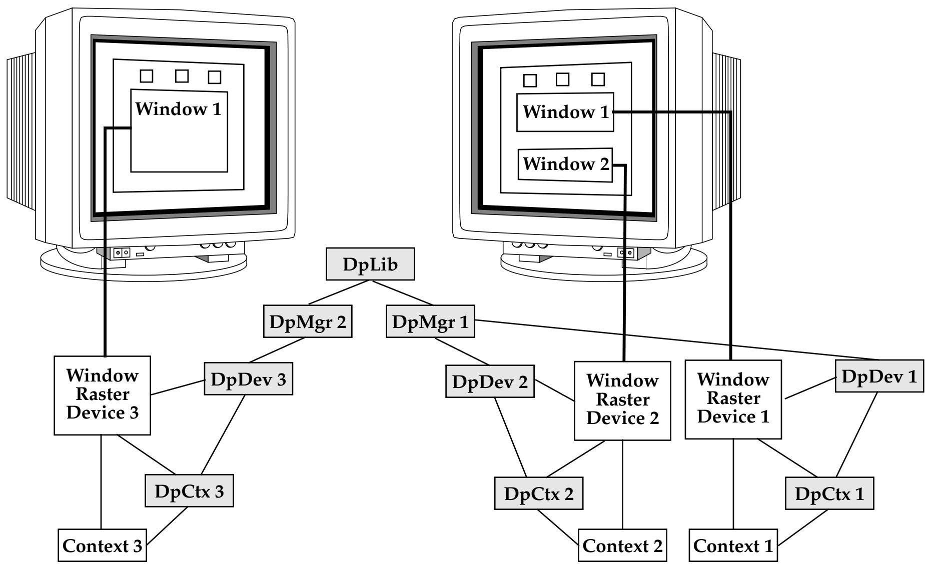


Figure 2.3 XGL Pipeline Objects

XGL Integration with OpenWindows

An XGL application is an X client program running in an X window system environment such as SunSoft's Solaris OpenWindows environment. The X window system is made up of several components. The most important of these is the X server, which is responsible for managing output requests from client applications to draw on the display and for distributing input events to the appropriate client applications.

Client applications and the X window server communicate via the X11 protocol. During a work session, client applications generate protocol queries and requests using the Xlib programming interface or an X toolkit; the server processes these requests.

X applications create, use, and destroy resources during their execution. A resource is an abstraction of a complex entity, such as a window or a pixmap. Resources allow applications to interact with the physical features of a display and to use state information about the display, such as window geometry and colormap.

An application program using XGL uses Xlib or X toolkit functions to connect to the server, create the window on the display, and handle event interpretation. The Xlib or X toolkit calls coexist with the XGL calls, sharing the same process space and drawable area on the screen. Xlib or the X toolkit manage the relationship among the windows, and provide input to application. An X toolkit such as the Solaris OPEN LOOK Intrinsics Toolkit (OLIT) also provides an application with a set of user interface mechanisms that ensures a standard 'look and feel' across applications using the same toolkit.

OpenWindows Server Capabilities

XGL utilizes SunSoft's Direct Graphics Access™ (DGA) technology to directly render via hardware accelerators whenever an application is running locally. The X interprocess communication mechanism is bypassed to permit high-performance rendering and a more direct mapping of functionality between the graphics library and the accelerator hardware.

The OpenWindows server also provides other graphics-related features, including:

- Multiple visual type support, including 4- and 8-bit PseudoColor, StaticColor, GrayScale, StaticGray, DirectColor and TrueColor (on 24-bit devices)
- Access to hardware double buffering
- Support for additional peripherals, such as buttons, dial boxes, joysticks, etc.

Direct Graphics Access™ (DGA)

SunSoft designed DGA into the OpenWindows server to enable locally rendered graphics to fully utilize the functionality and performance of graphics accelerated framebuffers and processors. XGL uses DGA to communicate and synchronize with the X server, and thus maximize throughput between the graphics client application and the hardware. Currently, any API or application using XGL, XIL or Direct Xlib take advantage of DGA.

DGA also efficiently overcomes performance constraints associated with the Interprocess Communication (IPC) mechanism employed with the X protocol. The IPC bottleneck has inherent overhead associated with an extra copy operation between the client application, the network socket and the server.

The use of DGA is transparent to the application developer. Whenever an application runs on the same machine as the server, XGL detects it, and invokes DGA to coordinate display sharing between the client and the window system.

Running XGL Across the Network

Like all X applications, an XGL application can be run remotely and displayed on a user's local workstation, provided the two systems are part of a network. Remote rendering through the X11 protocol or the PEX protocol is handled automatically by XGL. However there may be performance implications associated with running XGL across a network. When a client application is running locally under SunSoft's Solaris OpenWindows server and the hardware has DGA support, XGL uses DGA technology to enhance performance.

Although the server manages the resources of the device, the XGL client process can coordinate with the server to send rendering commands directly to the device rather than sending X11 protocol messages to the server. This allows XGL to eliminate the communication overhead of the server-based window system.

Figure 2.4 illustrates XGL's relationship to DGA and the server. When an XGL application creates a Raster Device object, XGL automatically determines whether DGA is available and whether XGL supports the display device. If DGA is available and the device is supported, XGL synchronizes with the server to share direct access to the display. In this case, XGL renders graphics geometry to the display device, while the server continues to perform other window operations.

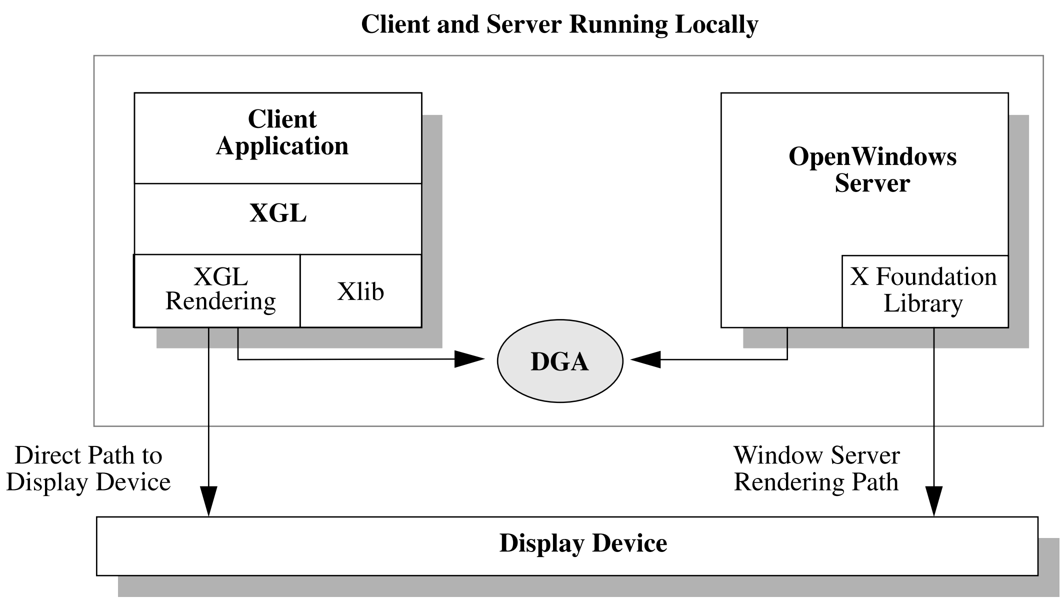


Figure 2.4 DGA and the OpenWindows Environment

When the client program is running remotely, meaning the client and the server are running on different machines, XGL uses Xlib or PEXlib to do all rendering. If the server includes the PEX extension and XGL can use a PEX loadable pipeline. This special kind of pipeline uses PEXlib to do 3-D rendering and Xlib to do 2-D rendering. If PEX is not available, XGL calls Xlib via an Xlib device pipeline. Figure 2.5 illustrates remote rendering of an XGL client application.

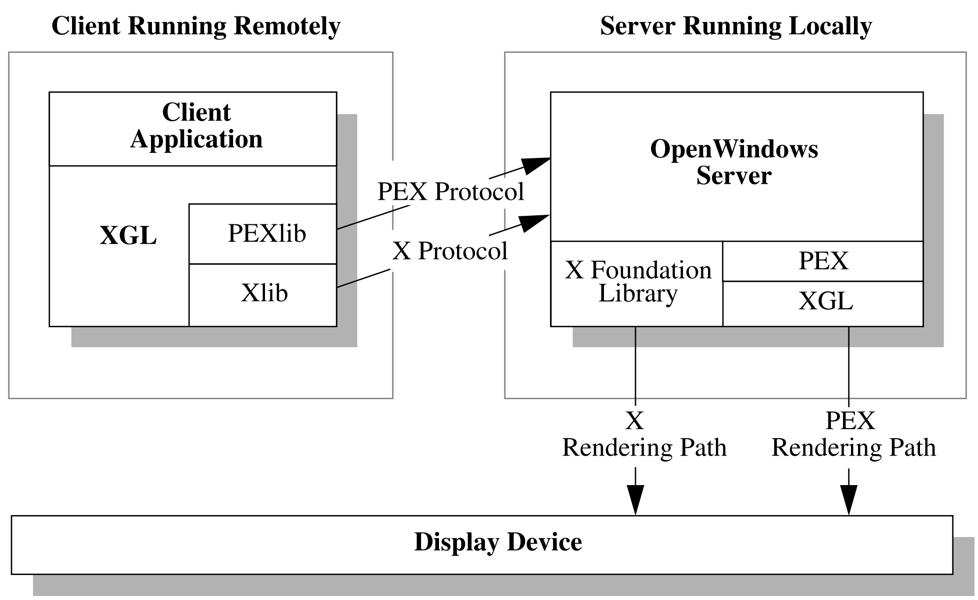


Figure 2.5 X/PEX Protocols and the X Windows Environment

Graphics Interface Interoperability and Independence

Any Solaris graphics API may render to the same canvas as another, at the same time. However, no interface is specifically aware of the rendering that other interfaces have done to the canvas, and will not assume responsibility to re-display. The application is responsible for repaints in response to X events. The application calls XGL to redisplay when necessary.

Application programmers which call more than one interface, such as XGL and Xlib, should be careful when mixing calls from both interfaces. Latencies involved in communication between high-level interfaces, graphics hardware and the window server make it possible for a primitive drawn by one interface to appear out of order to a primitive drawn by another interface. If the order of the primitives to be rendered is important, the application needs to assume responsibility for synchronizing among the multiple interfaces into an X window.

Toolkit and Window Manager Considerations

Toolkit independence permits the developer more freedom to select a Graphical User Interface toolkit, appropriate to his application, functionality and porting needs.

Virtually all graphics libraries available under Solaris render to an X11 Window, hence any toolkit that uses X can support applications through these interfaces.

Concurrent rendering from multiple graphics APIs to a single canvas is also possible. In this situation, the application must manage rendering independently for every API; each has no knowledge of the others presence.

To take advantage of DGA performance, it is important that the toolkit selected run in the OpenWindows environment on top of the OpenWindows server. Enhancements have been made to the server which DGA requires.

SunSoft provides several toolkits as part of its Solaris OpenWindows distribution. The OPEN LOOK Intrinsics Toolkit (OLIT) provides a common look and feel (OPEN LOOK) for applications running under OpenWindows. Third-party toolkits running with OpenWindows are also available including several implementations of the Motif toolkit. Provided the toolkit runs with OpenWindows, XGL can be used to render to the X Window canvas, and will utilize DGA when rendering locally.

Like toolkit independence, Window Manager independence gives application developers the ability to select whatever X-compliant window manager they choose.

SunSoft provides the OPEN LOOK Window Manager (olwm) with its OpenWindows environment. Using facilities such as drag-and-drop under OpenWindows requires the olwm window manager. Third-party window managers running with the OpenWindows server and supporting OPEN LOOK (OLIT and XView) and other toolkits are also available and work with both XGL and DGA.

The XGL Core Library 3

Graphical Organization

XGL is structured hierarchically through a series of abstract data types called *classes*. 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 which operators can act on it. A class defined hierarchically below an existing one is known as a *subclass* (child) of the existing (parent) class. Although XGL is class-based, an application can only define instances of predefined classes (*objects*), and cannot extend the hierarchy by creating new classes. Figure 3.1 illustrates XGL's Class hierarchy.

XGL Objects

In XGL, an *object* is an abstraction of a graphics resource. Objects consist of *state information* (known as attributes in XGL) and a set of operators. Operators perform predefined actions on objects of their related class.

The primary XGL objects available for the manipulation of 2-D and 3-D data are *System State*, *Context*, *Device*, *Gcache*, *Color Map*, *Dmap Texture*, *MipMap Texture*, *Transform*, *Stroke Font*, *Line Pattern*, *Light*, and *Markers*. The following paragraphs provide a brief discussion of XGL objects.

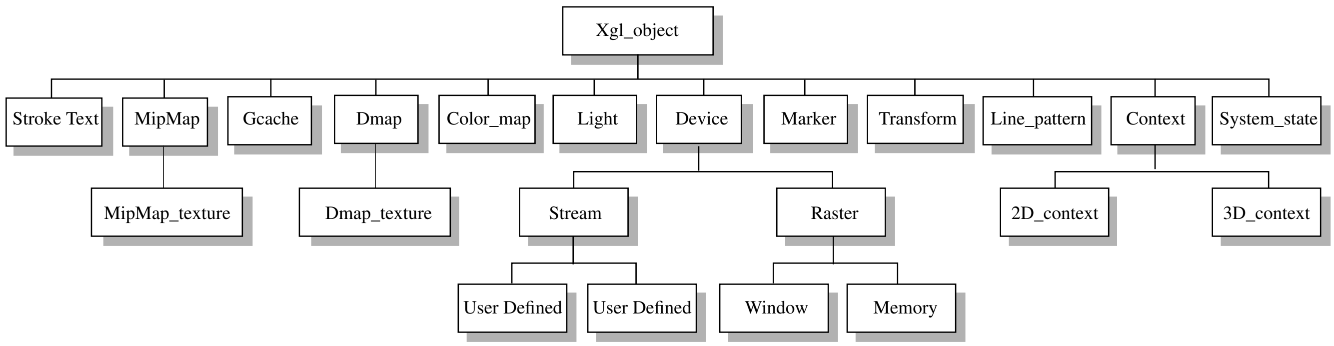


Figure 3.1 XGL Class Hierarchy

System State Object and Generic Operators

The XGL System State object maintains information pertaining to all operations occurring during a single XGL session. Operators of the System State object open and close XGL and adjust the degree of error checking XGL performs. The open and close operators, respectively, are always the first and last XGL operators executed from within an application.

The 2D and 3D Context Objects

The 2D and 3D Context objects provide 2-D and 3-D drawing primitives and graphics attributes for basic rendering. They are the most complex XGL objects. Several Context objects can exist simultaneously and can share resources associated with other XGL objects. 2D Context and 3D Context objects are separate, distinct objects designed to meet the dual goals of performance and dimension.

The Context object maintains both graphic and environment information on Coordinate Systems. The View Model, part of the environment information, performs general and specialized transformation and clipping operations, such as for lines and markers, 2-D and 3-D curve and surface tessellation, and stroke text rendering.

The Device Object

The XGL Device object retains information directly related to the drawing surface. The XGL Device object can be either a raster or non-raster device. If the device belongs to the raster class, it can be either a *Window Raster* (displayed on a screen) or a *Memory Raster* (consisting of an off-screen array stored in system memory).

Stream devices provide protocol-independent stream-I/O pipelines, for creating formatted output such as PostScript or Computer Graphics Metafile (CGM). The XGL Stream device maintains no protocol itself, but instead relies on an instantiated user or system-defined non-raster device pipeline to implement the appropriate output protocol data format.

Device objects can be attached to one or more Context objects, as the destination of drawing operations executed on the Context(s).

The Window and Memory Raster Devices are configured using the Raster Device attributes. For Window Raster Devices, the Device object maintains the connection to the windowing system, OpenWindows, or another X11 Windows server.

The Geometry Cache (Gcache) Object

The XGL Gcache Object reduces the complexity of application geometry by allowing XGL to process the geometry into simpler forms. It reduces single complex primitives into many simple primitives, storing the relevant attribute values. The Gcache object additionally provides a mechanism to store graphics primitives that are rendered frequently, thus yielding higher performance

The Color Map Object

XGL Color Map objects specify colors in a device-independent manner. The XGL application can work within its own color space, even if the hardware employs a different one. The Color Map object attached to the device ensures optimal correspondence between the application's color model and the color model of the device. XGL supports both indexed and RGB color models.

The Transform Object

The XGL Transform Object provides attributes and operators for geometric transformations performed on graphics primitives. These include translation, scaling, and rotation, as well as conversion from one coordinate system to another. A single 2D Context or 3D Context typically uses several Transform objects, which in turn can be shared (as defined by the application) between several Context objects.

The XGL Library supports both single and double precision floating point representations for applications requiring very large dynamic range or extra precision for transform operations.

The Stroke Font Object

The XGL Library provides Stroke Font objects that may be shared between Contexts. An XGL application may use Stroke Fonts to render stroke text in a variety of font types on Devices attached via Context operations. The XGL Library provides applications with high-quality fonts through improved interspacing of characters. This has the effect of making fonts more readable, particularly when text is small.

The Line Pattern Object

The XGL Library provides a Line Pattern object that is sharable between Contexts. Line Pattern objects allow non-solid lines to be rendered on a Device. The application may explicitly define the Line Pattern or may use one of several predefined Line Patterns XGL creates internally.

The Light Object

The Light object allows a 3D Context to render graphical primitives with *ambient*, *positional*, *directional*, and *spot* light sources. Light objects can be shared by different Contexts while remaining independently controllable on a per-context basis. This provides flexibility to the application, permitting individual Contexts to share some lighting attributes (such as position) but not others (such as light types enabled).

The Marker Object

The Marker object allows applications to describe their own markers or symbols and share them between different Contexts. A user-defined Marker can contain as many as 128 vectors.

The Dmap Texture Object

The Dmap Texture object allows an application to specify the control parameters needed to do texturing of 3-D surface primitives when rendering through a 3D Context. A Dmap Texture object can have many MipMap Texture objects and each 3D Context can have many Dmap Texture objects.

The MipMap Texture Object

The MipMap Texture object allows the application to specify texture images, which are generally predefined raster images. The MipMap Texture object also provides filtering capabilities to generate MipMaps.

How the XGL Programming Interface Works

When an application opens the XGL Library, XGL automatically creates the System State object and other internal objects that handle interactions between the loadable pipelines and the device independent parts of XGL. To set up a framework for rendering, the application must create a Context object, which controls all rendering actions on an associated device, and a Device object, which enables a Context object to render geometric primitives to a drawing surface. These objects are created using the *xgl_object_create()* operator.

The *xgl_object_create()* operator creates an instance of an XGL class. It takes attributes describing the object as arguments and returns a handle to the object. Before geometry can be rendered, the application program must associate the Context object and the Device object. These objects are associated by assigning the handle of the Device object to a Context object attribute, either when the Context object is created or at any time before rendering occurs. Other objects, such as multiple Stroke Font objects or new Line Pattern objects, can be created as needed by the application and then associated with the Context object. Multiple Device objects, such as Window Raster and Memory Raster devices can be associated and disassociated with the Context object as needed for rendering. Figure 3.2 illustrates object instantiation through the XGL Library.

Other relationships are managed internally by XGL object management functions. When an application has created 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. Figure 3.3 lists objects that are associated with the Context and Device objects as graphics resources.

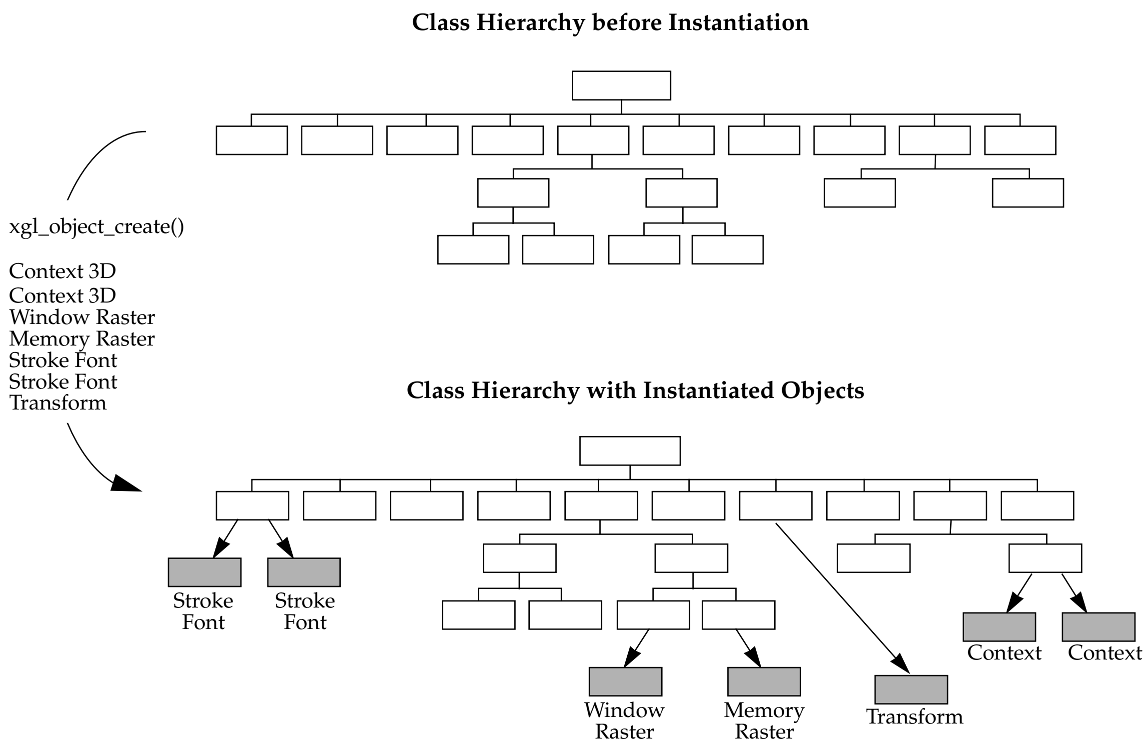


Figure 3.2 XGL Object Instantiation

User Object	**Related Object (Resources)**
Raster Device	System State Color Map
Context	System State Device Transform Line Pattern Marker Stroke Font Light (3D Context only) DMap_texture (3D Context only)

Figure 3.3 XGL User Objects and Related Objects

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 (VDC) 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. Each attribute-value pair in the list consists of an attribute name followed by its value. The attribute-value list may have as few as zero attribute-value pairs or as many as the total number of the object's possible attributes. NULL is used to terminate the attribute-value list.

XGL Operators

The *xgl_object_create()* operator is one of a set of operators that XGL uses to create objects and to modify the attributes of objects. XGL includes three types of operators: generic operators used with all API objects, geometry rendering operators, and object-specific operators. The generic operators, listed in Figure 3.4, provide a consistent method of working with all XGL objects.

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

Figure 3.4 Generic XGL Operators

The rendering operators are the XGL drawing primitives. XGL provides a broad range of drawing primitives, including polylines, text, and filled areas such as rectangles, quadrilateral meshes, multiple single-boundary polygons, and multiple-boundary polygons. Drawing primitives are available to an application as operators of the Context object. At execution time, a primitive gets the attributes it needs by rendering from the Context object. Figure 4.1 illustrates the XGL rendering operators and their associate primitives.

XGL Features and Description 4

Graphical Display

The XGL interface renders onto the graphical display through the 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 when the raster is created with the generic XGL creation operator *xgl_object_create()*.

The XGL Library requires a window system in which to run when rendering to a framebuffer or other raster graphics device associated witha CRT screen; it does not work with a raw screen. However, rednering to a memory raster or system device does not require a window system. An XGL application may also utilize hardware double-buffering capability when it is available. A flexible buffer-switching mechanism is provided as part of the implementation.

Primitives

The XGL Library accepts a large number of different representations of geometrical data, providing applications with flexibility in choosing an appropriate implementation. Data representation includes coordinate type and attributes associated with the rendered geometry. Figure 4.1 illustrates the most commonly used XGL primitives (rendering operators) and some of their associated attributes. Most primitives are renderable through both 2-D and 3-D pipelines.

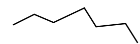
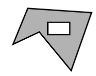
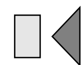
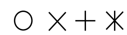
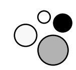
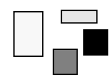
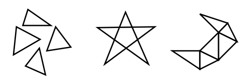
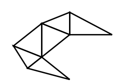
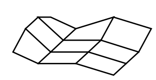
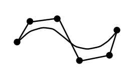
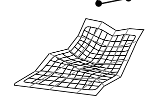
Rendering Operator	Description	Attributes	Example
xgl_multipolyline	Set of connected lines	Line Color, Style, Width	
xgl_polygon	Polygons	Polygon Color Fill Pattern Edge Style, Width Illumination Type	
xgl_multi_simple_polygon	Polygons (single boundary, 3 or 4 vertices)	Polygon Color Fill Pattern Edge Style, Width Illumination Type	
xgl_multimarker	Marker symbols	Marker Color Scale Factor	
xgl_multiarc	Arc	Arc Color, Style, Width	
xgl_multi_elliptical_arc	Arc	Arc Color, Style, Width	
xgl_multicircle	Circle	Circle Color Edge Style, Width Fill Pattern	
xgl_multirectangle	Rectangle	Rectangle Color Edge Style, Width Fill Pattern Illumination Type	
xgl_triangle_list	Series of Triangles (via a list) (triangle strip, triangle star or independent triangles)	Triangle Color Fill Pattern Edge Style, Width	
xgl_triangle_strip	Triangle strip (interconnected triangles)	Triangle Color Fill Pattern Edge Style, Width	
xgl_ quadrilateral_mesh	Quadrilateral mesh (interconnected quads)	Quad Color Edge Style, Width Fill pattern Illumination Type	
xgl_nurbs_curve	Non-Uniform Rational B-Spline Curve	Line Color Line Style, Width	
xgl_nurbs_surface	Non-Uniform Rational B-Spline Surface	Surface Color Edge Style, Width Fill Pattern Illumination Type	
xgl_stroke_text	Stroke text	Text Font Name, Color Character Size, Width Character Direction	TEXT

Figure 4.1 XGL Rendering Operators

The XGL interface includes several advanced primitives such as triangle strips, quadrilateral meshes, Non-Uniform Rational B-Spline (NURBS) curves and surfaces with dynamic tessellation, and annotation text functions. Such primitives are a powerful complement to the simpler primitives because they allow data compaction, utilize optimal clipping algorithms, and provide the best possible hardware rendering performance.

Transformations

Geometric transformations specify a linear mapping of one coordinate space to another. A transformation can be applied to the x,y, and z coordinates of geometric primitives using Transform objects. Included in the graphics state of a Context are various modeling, viewing, and windowing transformations, which are defined by transform objects in a view model. Some graphics accelerators have a hardware matrix multiplier and storage for one or more transformation matrices. The XGL Library caches transform objects within the graphics accelerator whenever possible.

Any linear transformation of coordinates (e.g. scaling, translation, or rotation) can be described as a sequence of 3-D matrix operations. A sequence of operations can be concatenated to form a single transformation that embodies all operations in combination. Transforming any point by a concatenated transformation produces the same result as applying the original transformations in sequence.

Like coordinates, transform objects may be specified as single or double precision floating point or 32-bit integer. Data may be associated with a transform explicitly by directly loading a 4x4 matrix for a 3-D transform, or a 3x2 matrix for a 2-D transform. An application can also specify the contents of a transform object in terms of a sequence of rotations, translations, and scalings. Figure 4.2 illustrates the common set of transformations applied to geometrical data.

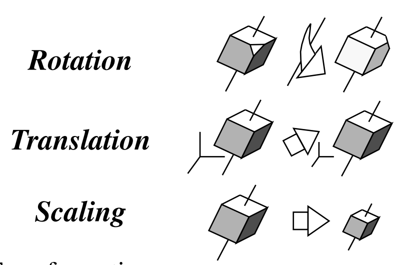


Figure 4.2 XGL Transformations

Color Models

A colormap object is used to define color value conversions from one color space to another. The conversion is done using a color table or a color mapping function. The color table is generally used for mapping an index into a color value in the true-color or RGB (red, green, and blue) space, and the color mapping function is employed when mapping the color values from true-color space to an index.

Since colormaps are XGL objects, they can be shared among several XGL rasters. The current version of XGL supports indexed, gray, and RGB color models. In the indexed color model, indexes are mapped to RGB settings through integer offsets into a color map. The mapping is defined by a table of RGB values that the application constructs and passes to XGL.

The gray color model requires a single floating point value between 0 and 1 to define the intensity value.

The RGB color model requires that the application specify RGB settings directly as the intensity of the three primary colors (red, green, and blue). The application indicates the intensity value of each primary as a floating-point number between zero and one. Depending upon the intensity value of each primary, the application can represent any color that is a combination of the primaries.

An XGL application associates a colormap with a device (raster). When the colormap is created, the XGL graphics library uses a default colormap with two indexes. All raster devices are created with this default. If the device does not support true color, and the application is using a RGB color model, the XGL graphics interface simulates the specified RGB color as closely as possible.

Figure 4.3 illustrates the XGL color models.

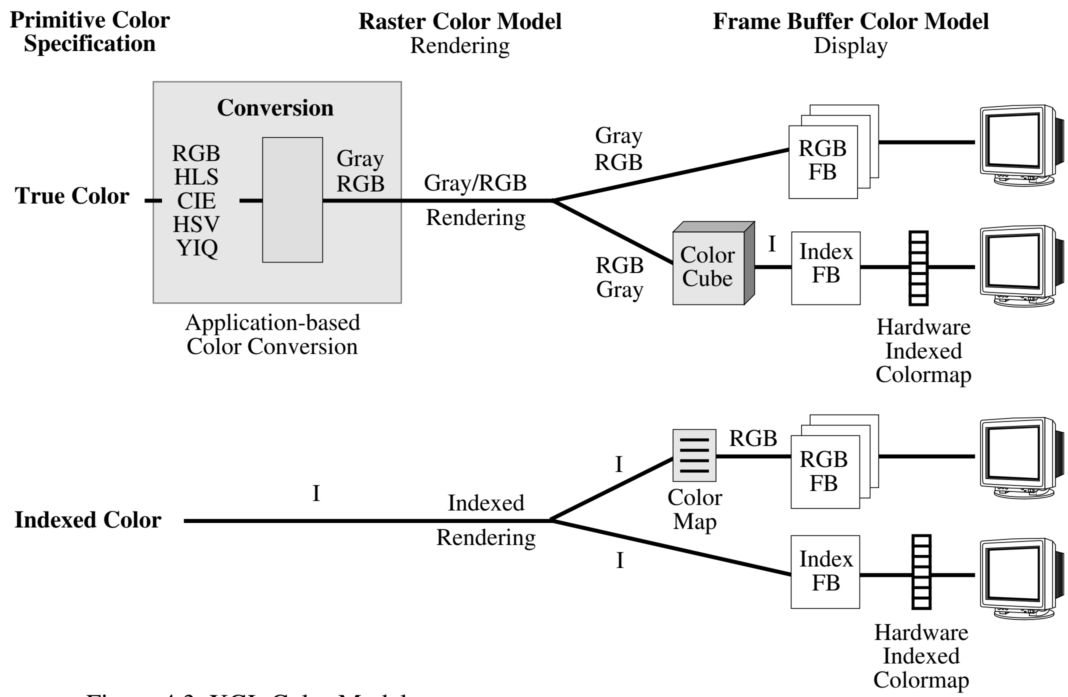


Figure 4.3 XGL Color Models

Lighting and Shading

The XGL interface supports flat and Gouraud shading. Lights are XGL objects and can be ambient, directional, positional, or spot. Lighting computations utilize the ambient, diffuse, and specular components to produce more realistic rendering. For true-color rendering, the RGB components of the light and object color are used for lighting and shading computations.

The XGL interface introduces a special element to ensure correct computations with an index model: *color_map ramps*. Users can define a set of ramps of different colors in the color map. The XGL Library automatically uses these ramps to evaluate the lighting and shading, while performing all computations in index mode. This reduces by a factor of three the number of computations involved in lighting and shading when rendering on index rasters.

Advanced Features

XGL includes advanced features which provide powerful capabilities to 3-D MCAD, animation and simulation applications. These features include advanced primitives such as triangle strips, quadrilateral meshes and NURBS curves and surfaces. Additionally, XGL provides support for depth cueing, transparency, antialiasing, accumulation buffers, and texture mapping of surface primitives.

XGL additionally includes a *triangle list* primitive to provide flexibility for applications rendering triangle-based geometry where the triangle data must be unconstrained in terms of vertex sharing.

Strips and meshes are optimized to give the fastest possible rendering speed. The XGL Library provides users with an optimal clipping method for triangle strips that takes advantage of the intrinsic structure of the primitive while continuing to maintain this structure after clipping. This approach limits the number of computations to a strict minimum, yielding maximum performance in rendering. Quadrilateral meshes are not restricted to planar geometry and are rendered row by row, using the triangle strip pipeline. In this manner, they can take advantage of the same optimizations as triangle strips.

NURBS (Non-Uniform Rational B-Spline) Curves and Surfaces

NURBS curves and surfaces are concise, yet powerful and general representations of a wide variety of simple and complex geometries. Trimmed NURBS are becoming increasingly more prominent in both informal and formal industry standards for geometric modeling and computer graphics. NURBS are effective for representing both simple and complex shapes, and include powerful mathematical properties which provide ease of manipulation and processing.

From a high-level mathematical description, a graphics system tessellates curved geometry into lines (in the case of curves) or triangles (in the case of surfaces), whose sizes depend on the spatial resolution of the curve or surface to the eye, and on the complexity of the geometry.

XGL NURBS curves are general enough for users to generate uniform curves such as Bezier curves, or uniform B-Splines. The order of the curves can be up to twenty-five (most CAD applications use three or four). XGL attributes offer complete control of the precision associated with the curve rendering such as dynamic tessellation to optimize the trade-off between number of points computed and rendering time. The XGL Library also offers a constant subdivision attribute to control the distance between two consecutive knots.

Accumulation Buffer

The XGL Raster object can provide antialiasing of images through the use of an accumulation buffer. An accumulation buffer is a buffer containing an array of pixels where the contents of the accumulation process is stored. The accumulation process takes a series of images (from multiple renderings) and combines them together.

With each successive rendering, the resulting image is moved or jittered a subpixel distance relative to the previous pass. It is then combined with the previously rendered images in the accumulation buffer. The combining of images in the accumulation buffer takes place through the weighted addition of the rendered image. The resulting image is effectively antialiased. XGL calculates the resulting contents of the accumulation buffer with the formula:

(destination buffer) =
*(source buffer * source weight + destination buffer * accumulation buffer weight)*

Typically, an application program writes to the accumulation buffer multiple times to produce a desired image, and this image is then copied to the display buffer. Accumulation buffering can also be used to render motion blur, depth of field or soft shadows.

Applications requiring an accumulation buffer must use the *xgl_context_accumulate* operator. Figure 4.4 lists the accumulation buffer attributes associated with the XGL primitive call.

Attribute	**Description**
XGL_3D_CTX_ACCUM_OP_DEST	Specifies the buffer to be used as the destination buffer.
XGL_RAS_ACCUM_BUFFER_DEPTH	A read-only attribute identifying the depth of the accumulation buffer.
XGL_3D_JITTER_OFFSET	A 3D Context attribute which specifies the amount by which the geometry should be offset in DC space.

Figure 4.4 XGL Accumulation Buffer Attributes

Geometry Cache (Gcache)

XGL provides a basic buffering mechanism to cache the geometric representation of data that will be rendered repeatedly. Known as Gcache, it is an XGL object that caches the underlying geometry for a particular primitive. For example, a Gcache could contain the polylines that comprise a string of stroke text, or triangles that make up a polygon.

A Gcache can be used to break down complex primitives into simpler forms, resulting in increased rendering performance (assuming the underlying accelerator is optimized for the resulting decomposed primitive). A Gcache also enables an application to instruct XGL to model clip, or decompose a primitive once, and then render the result many times.

Once a Gcache object is created, an application can use Gcache operators to store a primitive into the Gcache. For example, *xgl_gcache_polygon()*, which is used to store a polygon in a Gcache, is the Gcache counterpart of *xgl_polygon()*. The primitive can be displayed repeatedly, usually with better performance than its non-Gcache counterpart, by calling *xgl_context_display_gcache()*.

Whenever an object is stored in a Gcache, certain Context attributes (which depend upon the primitive being stored) will be saved in the Gcache. If a primitive is stored in a Gcache, and any relevant Context attribute changes, the properties of the cached primitive will be outdated. When this happens, the application can decide if the primitive should still be displayed. Many Context attributes can be changed after a primitive has been stored in a Gcache without outdating the primitive. For example, view clipping attributes can be changed and the primitive will still be rendered correctly.

Many graphics hardware accelerators are efficient with simple geometries. For example, an accelerator may draw triangles very fast. If an application wants to render the same complex polygon several times, it is worthwhile to use a Gcache to break the polygon into triangles (known as tessellation) and send the triangles to the accelerator for rendering. If a Gcache was not used, then tessellation would have to be done each time the polygon was rendered.

For stroke text, a Gcache uses the character attributes (character spacing, height, etc.) to convert the text to a set of polylines for faster rendering. For NURBS curves and surfaces, much of the processing time is absorbed during Gcache creation time. This substantially speeds up display of NURBS curves and surfaces. Figure 4.5 illustrates the creation and manipulation of NURBS primitives using a Gcache.

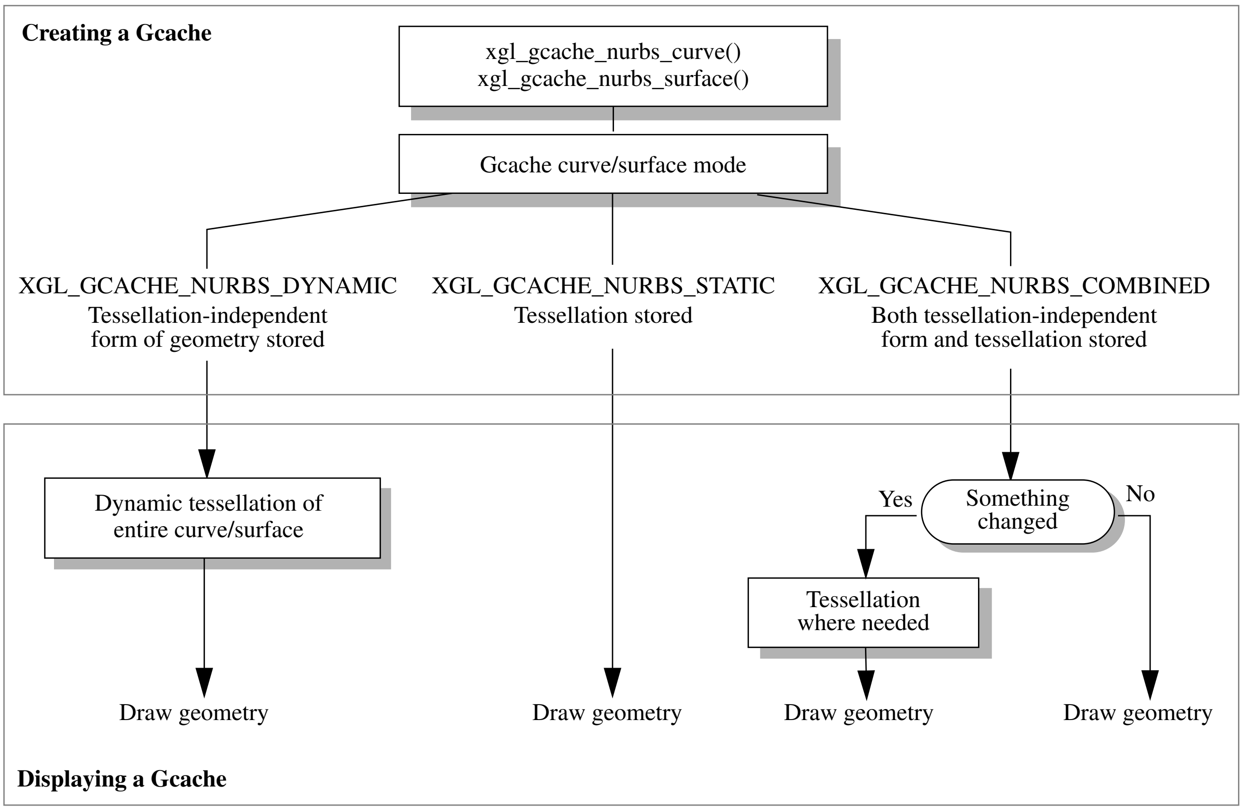


Figure 4.5 Using XGL's Gcache for Rendering NURBS Curves or Surfaces

Transparency

XGL supports several transparency methods for 3-D surfaces, including screen-door transparency and blended (two-pass) transparency. Transparency can be invoked through 3D Context attributes assigned to the primitive to be rendered.

Screen door transparency allows a device-dependent screen door to render the object using a mesh through which some of the primitives pixels are rendered. If the blended method is used, blending equations determine how the surface pixel values are blended with the background color or existing pixel values.

Transparency may be achieved in a number of ways, dependent upon available display hardware. For example, transparency may be directly supported on some devices or may use stipple patterns on others.

Transparent surfaces are subject to hidden line and surface removal as defined by the XGL_3D_CTX_HLHSR_MODE attribute. Figure 4.6 lists the surface transparency attributes that can be assigned to a primitive prior to rendering.

Attribute	Description
XGL_3D_CTX_SURF_TRANSP_METHOD	Defines the method used to render transparent surfaces.
XGL_3D_CTX_SURF_FRONT_TRANSP XGL_3D_CTX_SURF_BACK_TRANSP	Specifies the transparency value for surface primitives. Surface transparency can range from fully transparent through translucent to opaque.
XGL_3D_CTX_SURF_TRANSP_BLEND_EQ	Controls how the pixel values in transparent surfaces are blended with existing pixel values when the surface transparency method is XGL_TRANSP_BLENDED.
XGL_3D_CTX_BLEND_DRAW_MODE	Controls the drawing of blended and non-blended primitives.
XGL_3D_CTX_BLEND_FREEZE_ZBUFFER	Defines whether blended primitives update the Z-buffer.

Figure 4.6 XGL Transparency Attributes

Depth Cueing

Depth cueing assists observers with perception of depth by fading the primitive's colors by an amount that corresponds with the distance of the primitive from the observer. Typically the amount of fading increases with the distance.

Applications that employ depth cueing often combine this effect with motion through animation. Two common applications which use depth cueing are molecular modeling and flight simulation. In the case of molecular modeling, the background color is usually black. Depth perception can be enhanced by fading the more distant parts of the molecule to black, making them appear dimmer. Oscillation of the molecule about a vertical axis enhances the depth perception of the object.

XGL supports two methods for depth-cueing known as *linear* and *scaled*. In linear depth-cueing, the colors of the primitives are the same as the non-depth cued colors at the front of the Virtual Device Coordinate (VDC) viewport. At the back of the viewport, the colors completely change to the specified depth cue color (RGB) or the color at the bottom of the color ramp (for indexed color). Between these two boundaries, all colors modulate to a linear combination of the original color and the depth cue color.

Scaled depth-cueing is a superset of linear depth cueing. Scaled depth-cueing is linear between two predefined reference planes, which can be placed at arbitrary depths in virtual device coordinates. The scale factors for the front and back planes must fall within values between 0.0 and 1.0. The scale factors outside the linear region are clamped at the values of the reference planes closest to them.

Depth-cueing in XGL is achieved through the assignment of attributes for the primitive to be depth-cued. Figure 4.7 lists the depth-cueing attributes and their description.

Attribute	**Description**
XGL_3D_CTX_DEPTH_CUE_MODE	Defines the method used to depth cue: linear, scaled or off.
XGL_3D_CTX_DEPTH_CUE_INTERP	Specifies the interpolation method for determination of color.
XGL_3D_CTX_DEPTH_CUE_REF_PLANES	Defines two planes normal to the z-axis in virtual device coordinates (VDC).
XGL_3D_CTX_DEPTH_CUE_SCALE_FACTORS	Defines a portion of the primitive's color when the depth cue mode is set to XGL_DEPTH_CUE_SCALED.
XGL_3D_CTX_DEPTH_CUE_COLOR	Specifies the color that depth-cued primitives will tend toward for increasing values of Z.

Figure 4.7 XGL Depth-cueing Attributes

Texture Mapping

The XGL Library supports 2-D texturing of 3-D surface primitives using a texturing raster image specified by the MipMap_texture object. Textures are defined in a normalized texture space {u,v}, with both the u and v coordinates defined in the range 0.0 to 1.0.

Mapping of the texture onto a polygon is accomplished by listing {u,v} values in the 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}. Figure 4.8 illustrates the texture mapping space and the example described here. Figures 4.9 and 4.10 list the DMap and MipMap object texture attributes respectively.

Texturing is achieved in XGL through attributes which allow a great deal of flexibility. XGL allows the result of the texturing operation to be applied to different stages of the rendering pipeline (see Figure 4.11). It also supports sampling methods such as point, bilinear and trilinear to obtain texture value and several color composition techniques such as blend, decal and modulate. XGL additionally has the capability of applying several textures to individual primitives.

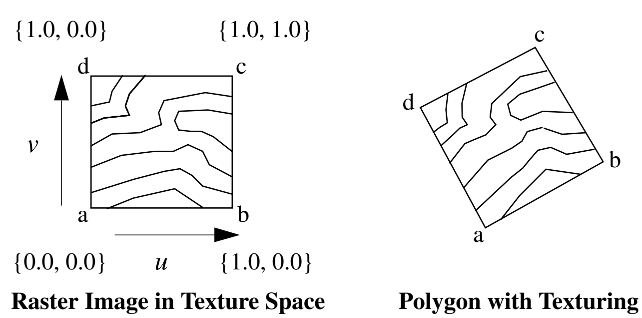


Figure 4.8 Texture Mapping of a MipMap to a Polygon

Attribute	Description
XGL_3D_CTX_SURF_FRONT_DMAP_NUM	Specifies the number of data map texture object for front face.
XGL_3D_CTX_SURF_BACK_DMAP_NUM	Specifies the number of data map texture object for back face
XGL_3D_CTX_SURF_FRONT_DMAP	Specifies a list of data map texture objects for front faces.
XGL_3D_CTX_SURF_BACK_DMAP	Specifies a list of data map texture objects for back faces.
XGL_3D_CTX_SURF_FRONT_DMAP_SWITCHES	Specifies which of the data map texture objects should be applied for front faces.
XGL_3D_CTX_SURF_BACK_DMAP_SWITCHES	Specifies which of the data map texture objects should be applied for front faces.

Figure 4.9 XGL DMAP Object Texture Mapping Attributes

Attribute	Description
XGL_MIPMAP_TEXTURE_WIDTH	Specifies the width of a texture image.
XGL_MIPMAP_TEXTURE_DEPTH	Specifies the depth of a texture image.
XGL_MIPMAP_TEXTURE_HEIGHT	Specifies the height of a texture image.
XGL_MIPMAP_TEXTURE_LEVELS	Specifies the number of levels of a texture image.
XGL_MIPMAP_TEXTURE_MEM_RAS_LIST	Gets or sets MIPMAP list of memory rasters.

Figure 4.10 XGL MipMap Object Texture Mapping Attributes

The XGL Rendering and Transformation Pipeline

The XGL Library rendering and transformation pipeline manages the rendering and transformation of geometric data from the API to the underlying display hardware. From an application's perspective, the pipeline provides individual stages for the manipulation of geometric properties such as coordinate transformation, lighting, depth cueing, shading and transparency. Figure 4.11 illustrates XGL's rendering pipeline.

For flexibility and performance reasons, the XGL Library employs separate viewing pipelines for 2-D and 3-D graphics. Figure 4.12 shows both the 2-D and 3-D transformation pipelines. The 2-D viewing model is an *affine* model, whereas the 3-D viewing is a *homogeneous* model. A viewing model allows the application to describe the portion of a scene that should be displayed. To achieve this goal, a set of attributes define the transformations and clipping operations to perform between the application's space and the display space.

Like many graphics interfaces, XGL provides several local model transforms. The local transform can be used to map an element of the graphic scene with respect to the complete scene. The global model transform can be used to map the complete scene to an arbitrary world coordinate system. The view matrix typically represents the set of transformations done to map the users' viewing direction and perspective to the virtual device space.

A set of attributes, Virtual Device Coordinate (VDC) window, VDC mapping, VDC orientation, and Device Coordinate (DC) viewport specify the mapping from the VDC space to the display area.

Clipping is conceptually performed in three places; optional model clipping may take place in World Space, optional clipping may take place in VDC coordinates, and clipping to DC limits is always applied to ensure the correctness of the drawings in a window. The 3-D viewing pipeline has some extra steps due to its homogeneous characteristics. Figure 4.12 illustrates the stages of rendering for 3-D primitives. In this figure, local and global transforms are represented by a single model transform for compactness.

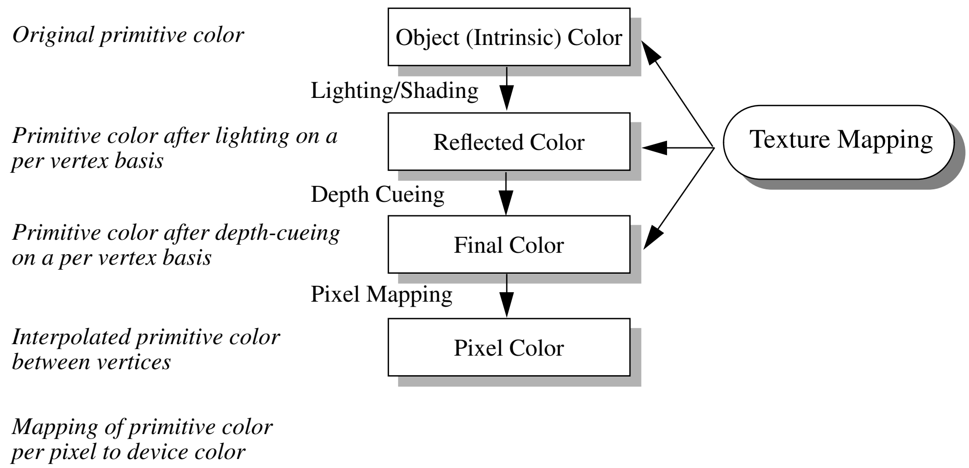


Figure 4.11 XGL Rendering Pipeline

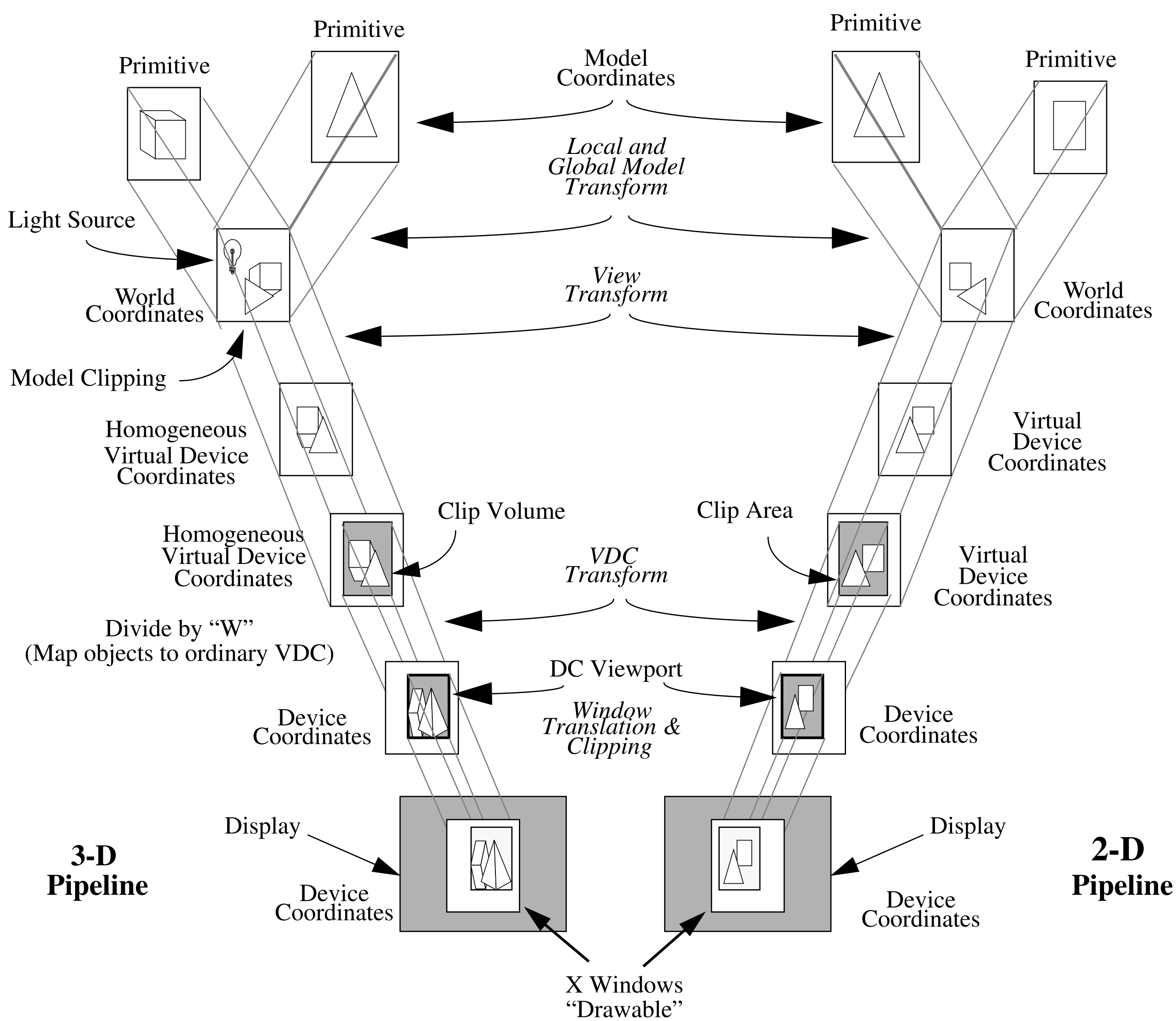


Figure 4.12 XGL 2-D and 3-D Transformation Pipelines

Text Font Support

XGL supports internationalized character sets through its stroke font object facilities. A string of text may be composed of distinct fonts built from different character sets. An XGL Context object offers two text encoding (Extended UNIX Coding and ISO) schemes and four attributes which are utilized to select up to four active, simultaneous character sets. Extended Unix Coding (EUC) defines an encoding for characters that allows multi-byte characters. EUC uses bit patterns and control characters to select different character sets within a string. Mono encoding represents a string as an array of ASCII elements.

The motivation for allowing different encoding schemes and up to four fonts per Context object is to support internationalization extensions by providing the ability to switch between different languages (character sets). XGL supplies a number of text fonts, as listed below.

Cartographic	Cartographic_M	English_G
Greek_M	Greek_C	Greek_M
Headline	Italic_C.phont	Italic_T
Miscellaneous	Miscellaneous_M	Roman
Roman_C	Roman_D	Roman_M
Roman_T	Script	Script_C

The fonts listed above include a '*.font*' extension as part of their name. Additional fonts to be utilized by XGL must be placed in the XGL font subdirectory ($XGLHOME/lib/xglfonts/stroke usually defined as /opt/SUNWits/Graphics-sw/xgl-3.0/lib/xglfonts/stroke)).

New XGL fonts may be created using the *Vector Font Editor* (vfe), which is provided as part of Developers Version of XGL. Vfe provides a visual interface for the modification of existing fonts or the creation of new ones. Vfe works with the extended ASCII representation of a font file. Fonts however are stored in a binary format, and must be converted to their extended ASCII representation prior to being edited. When font creation or modification is complete, the font is reconverted back into its native binary format.

Within an XGL application, text font attributes allow the control of character rendering. Figure 4.13 lists the text font attributes and their description.

In addition to normal stroke text, which is subject to orientation based upon transformations of the primitives relative to the viewer's position, XGL also supports annotated text. Annotated text maintains a constant orientation, typically parallel to the display surface. Annotated text is often useful in applications where text is an important descriptive part of the geometry being rendered. Examples include molecular modeling, MCAD, and flight simulation.

Attribute	Description
XGL_CTX_STEXT_CHAR_HEIGHT	Character height in local coordinates.
XGL_CTX_STEXT_CHAR_EXPANSION_FACTOR	Horizontal scaling factor for width of characters.
XGL_CTX_STEXT_CHAR_SPACING	Character spacing factor for gap between characters.
XGL_CTX_STEXT_CHAR_UP_VECTOR	Defines “up” direction for text rendering.
XGL_CTX_STEXT_CHAR_SLANT_ANGLE	Defines inclination of characters from “up” vector.
XGL_CTX_STEXT_PATH	Defines direction text will be rendered relative to “up” vector.
XGL_CTX_STEXT_ALIGN_HORIZ	Defines text alignment.
XGL_CTX_STEXT_COLOR	Specifies text color.
XGL_CTX_STEXT_PRECISION	Specifies minimum precision for rendering text.
XGL_CTX_STEXT_CHAR_ENCODING	Specifies which text encoding scheme is used.

Figure 4.13 XGL Context Text Attributes

References

XGL Reference Manual
XGL Programmer's Manual

OpenWindows User's Guide
OpenWindows Programmers Guide

X Protocol Reference Manual, O'Reilly & Associates. Inc.
PEX Protocol Specification, MIT

Xlib Reference Manual, O'Reilly & Associates. Inc.
*Xlib Programming Manua*l, O'Reilly & Associates. Inc.
An Introduction to Computer Graphics Concepts, Addison-Wesley

The RenderMan Companion, Upstill, Addison-Wesley

SunSoft, Inc
2550 Garcia Avenue
Mountain View, CA 94043