Introduction to XGL 1

Overview of XGL Functionality

XGL™ is a software library of two-dimensional (2D) and three-dimensional (3D) graphics primitive functions designed to support a wide variety of graphics-based applications. The XGL library provides immediate mode, nondisplay-list functionality and is suitable for supporting a number of graphics application programming interfaces (APIs), including GKS and PHIGS.

The XGL library implicitly uses hardware graphics acceleration whenever possible. Hardware graphics acceleration occurs automatically without explicitly requesting it from within an XGL program.

The XGL system requires a window system to manage drawing operations sent to a display device; it cannot render graphics on a raw display screen. An XGL application runs within a window environment managed by an X11-R5 compatible server, such as the X11 server within Sun's OpenWindows™ environment. XGL primitives can render images onto multiple graphical display devices that coexist within the same XGL session. An example of a display device is an X11 window linked to the window system with a window handle.

The XGL system insulates the application programmer from the specifics of the window system and the underlying hardware device with XGL *objects*, which describe virtual components of the graphics rendering system. XGL objects simplify the work of the application programmer by presenting a consistent, device-independent graphics model. For example, display devices, such as X11

windows, 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 XGL 2D Context or 3D Context objects. See Figure 1-3 on page 6 for a diagram of the XGL object hierarchy.

The XGL libraries reside directly above the hardware and firmware of the display device and graphics accelerator, minimizing software overhead within the XGL graphics rendering pipeline. XGL uses Sun's Direct Graphics Access (DGA) display technology to accelerate XGL applications. DGA arbitrates access to the display screen between an XGL Window Raster and the X11 window system. DGA is available for graphics accelerators in Sun's OpenWindows environment.

The XGL library provides the applications programmer with the following graphics capabilities:

Immediate-mode rendering
: Immediate-mode rendering refers to immediate rendering of graphics primitives without placing the data in a display list. XGL leaves organization of the data to the application, which permits efficient management of the graphics geometry without imposing constraints on how the geometry is stored or what specific information is kept in the display list. Immediate-mode rendering is preferable for highly interactive applications or for applications maintaining specialized data, such as those in electronic and mechanical CAD, animation, and simulation.

Loadable device pipelines
: XGL 3.0 and subsequent releases support loadable device pipelines. The loadable device pipeline architecture makes it easier for hardware developers who want to port the XGL library to their graphics devices. If a hardware vendor releases a new graphics device after a release of the XGL core library and provides a library supporting the loadable device interfaces for that device, an application compiled for XGL will work without recompilation on the new device. This feature allows graphics devices to be released independently of XGL releases.

Separate 2D and 3D graphics pipelines
: The XGL library supports separate, complete 2D and 3D rendering pipelines that are accessible within the same application. The 2D pipeline is optimized for 2D applications and does not carry the overhead of 3D functionality. The 3D pipeline has a rich set of capabilities for 3D applications. An application

can render 2D and 3D data to the same device. The XGL dual pipeline approach allows programmers to use either the speed of the 2D pipeline or the capabilities of the 3D pipeline.

Broad primitive and coordinate type support
The XGL library has a rich set of graphics primitives and view and modeling transforms. Standard features include 2D and 3D primitive support (such as lines and polygons); depth cueing, lighting, and shading; nonuniform B-spline curves and surfaces; and direct and indirect color model support.

NURBS surfaces
XGL supports nonuniform rational B-spline (NURBS) surfaces. This primitive provides computer-aided geometric design applications with curved surface support. The primitive follows the semantics defined by PHIGS PLUS.

Multi-primitive operators
The XGL library provides *multi-primitive* operators that permit batching of simple primitive calls into groups, based on knowledge of coordinate and attribute data. In applications such as electronic design, which repeatedly render the same type of primitive, the performance gains of batched primitive calls can be substantial.

Transparent acceleration
The XGL library provides direct mapping of graphics functionality to the underlying hardware. 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.

Xlib and PEXlib device support for distributed networks
The XGL system supports Xlib devices (X protocol) and PEXlib devices (PEX protocol), enabling XGL programs to run across a network on remote systems. If the server supports the PEX extension and the XGL system has access to the PEXlib loadable library, XGL will emit PEX protocol that the server can interpret to generate for 2D and 3D graphics on a remote display.

Toolkit independence
XGL works with OpenWindows-supported toolkits or window managers. OpenWindows provides full X Window System capabilities and includes support for the X11 server, the X toolkits OLIT and XView™, and the Xlib library. XGL also works with most X11-compatible toolkits.

1

The XGL design is best suited to applications that maintain their own display list or graphics data structure. By giving applications greater control over the graphics to be rendered on the device, the XGL library provides functionality not found in PHIGS or GKS.

An application can use the XGL library as a high-level graphics library that acts as the application's interface to the hardware, or it can use the XGL library as a foundation low-level library under another high-level graphics library, such as PHIGS, as illustrated in Figure 1-1.

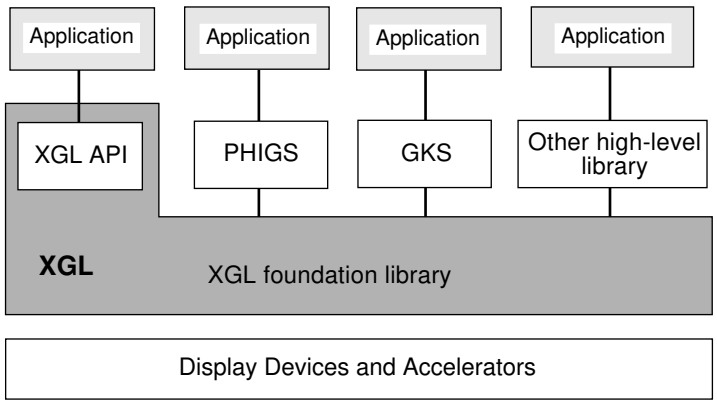

Figure 1-1 XGL API and Foundation Library

Direct Graphics Access

The XGL system uses Sun's Direct Graphics Access (DGA) technology to achieve high-performance graphics on graphics accelerators within an X11 window. As illustrated in Figure 1-2 on page 5, when a client XGL application is running locally (on the same machine as the X11 server), DGA allows the XGL library to send commands directly to the accelerator rather than sending protocol commands to the server. XGL thus avoids the overhead from building X11 protocol requests and passing them from client to server. Using a low-overhead synchronization mechanism, 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.

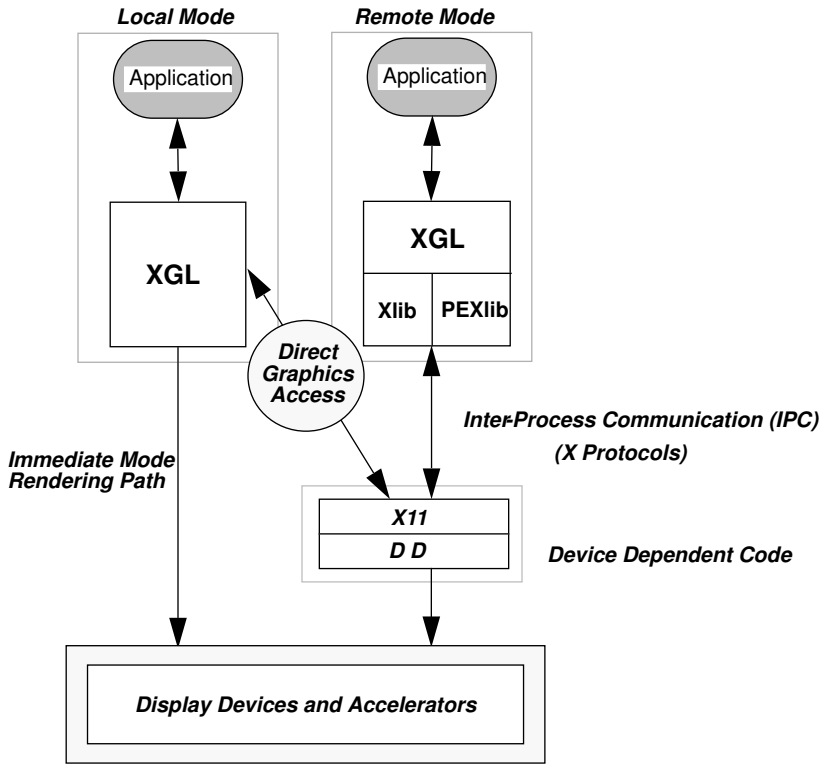

Figure 1-2 XGL in the OpenWindows Environment

Applications or new graphics interfaces layered on top of the XGL API automatically benefit from features implemented in the XGL library, such as DGA and remote rendering through X11 or PEX protocol.

1

Introduction to XGL Objects and Graphical Organization

The XGL API is structured hierarchically as a set of abstract data types called *classes* (see Figure 1-3). The class of an object determines the attributes it possesses and which operators can act on it. The class defined hierarchically below an existing class is a *subclass* (child) of the existing (parent) class. Applications can create *instances* of classes, where an instance of a class is an *object*. Although XGL is class-based, an application can only define instances of classes and cannot extend the hierarchy by creating new classes.

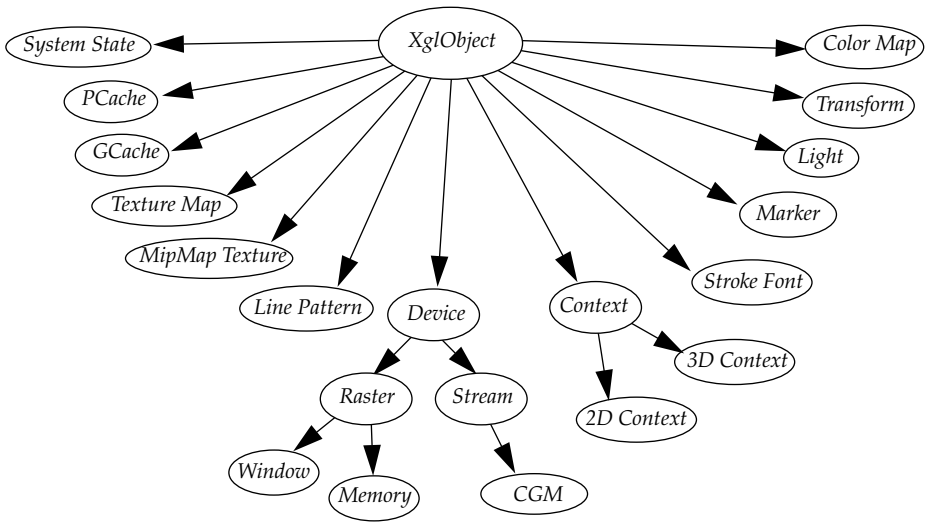

Figure 1-3 XGL Class Hierarchy

In the XGL API, 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 XGL objects available for the manipulation of 2D and 3D data are *System State*, *Context*, *Device*, *Pcache*, *Gcache*, *Color Map*, *Transform*, *Stroke Font*, *Line Pattern*, *Light*, *Marker*, *Texture Map*, and *MipMap Texture*. Following is a brief discussion of XGL objects; more information about each object type is provided in subsequent chapters.

System State Object and Generic Operators

The System State object maintains information about all operations occurring during a single XGL session. Operators of the System State object open and close XGL and adjust the degree of error handling that XGL performs. The open and close operators, respectively, are always the first and last operators executed from within an XGL application. Only one System State object is supported for any single XGL session. See Chapter 4, "System State Information and Generic Operators" on page 49 for more information on the System State object.

The generic operators create objects, retrieve attribute information about objects, change attribute values, and destroy objects. These operators are used with all XGL objects. Figure 1-4 illustrates the generic operators and their relationship to the application and the XGL runtime system.

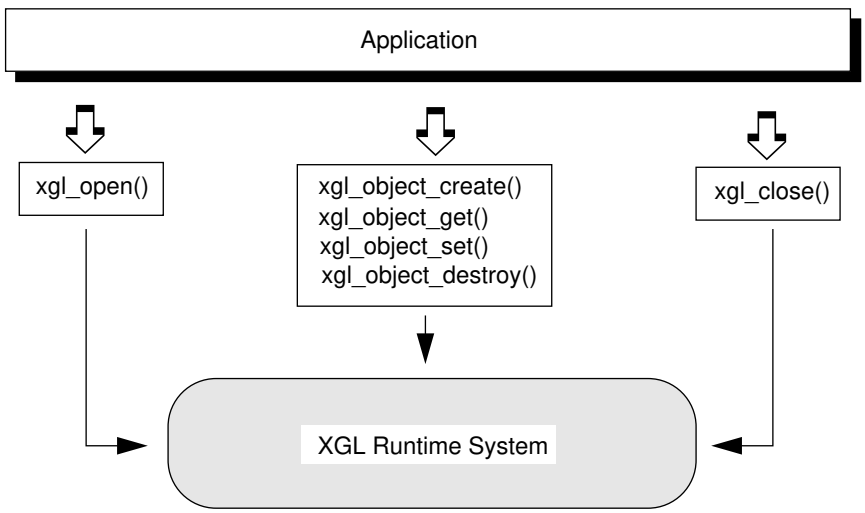

Figure 1-4 System State and Generic Operators

1

Device Object

The Device object maintains information directly related to the drawing surface and provides a framework for interacting with hardware graphics devices in an abstract manner. As shown in Figure 1-5, the Device class is subclassed to create a Window Raster object that renders onto a screen, a Memory Raster object consisting of an off-screen array stored in system memory, or a Stream object. A Stream device provides 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 the XGL-provided CGM device or an instantiated third-party non-raster device pipeline to implement the appropriate output protocol data format. The Device object can be attached to one or more Context objects, rendering the drawing operations executed from within the Context object on the attached device.

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. For more detailed information on XGL Device objects, see Chapter 5, “Devices” on page 59.

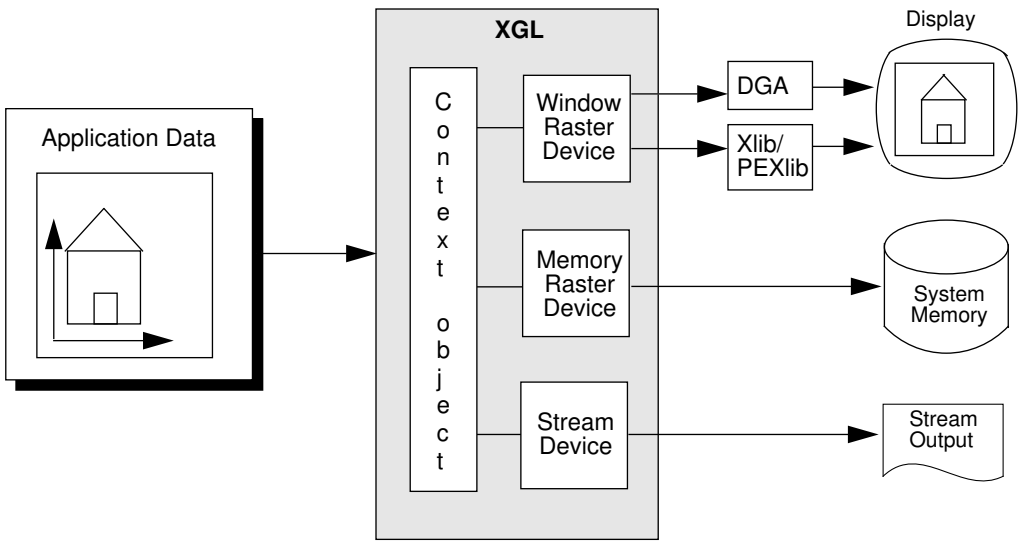

Figure 1-5 Overview of the Device Object

2D and 3D Context Objects

The 2D and 3D Context objects provide 2D and 3D drawing primitives and graphics attributes for rendering, as illustrated in Figure 1-6. The Context object also maintains graphic and environment information on coordinate systems and includes information that specifies the mapping and clipping of geometric data between coordinate spaces in the rendering pipeline. Several Context objects can exist simultaneously and can share other XGL objects as resources. 2D and 3D Context objects are distinct objects designed to meet the unique needs of 2D and 3D applications.

Primitives

XGL provides primitive operators as graphics tools for creating and manipulating geometric images. The XGL primitive operators are functions for drawing markers, polylines, curves, rectangles, arcs, circles, polygons, surfaces, and text. The characteristics of these primitives are specified in the Context object via Context attributes. For information on the Context object, see Chapter 7, “Contexts” on page 165 and Chapter 8, “Primitives and Graphics Context Attributes” on page 175.

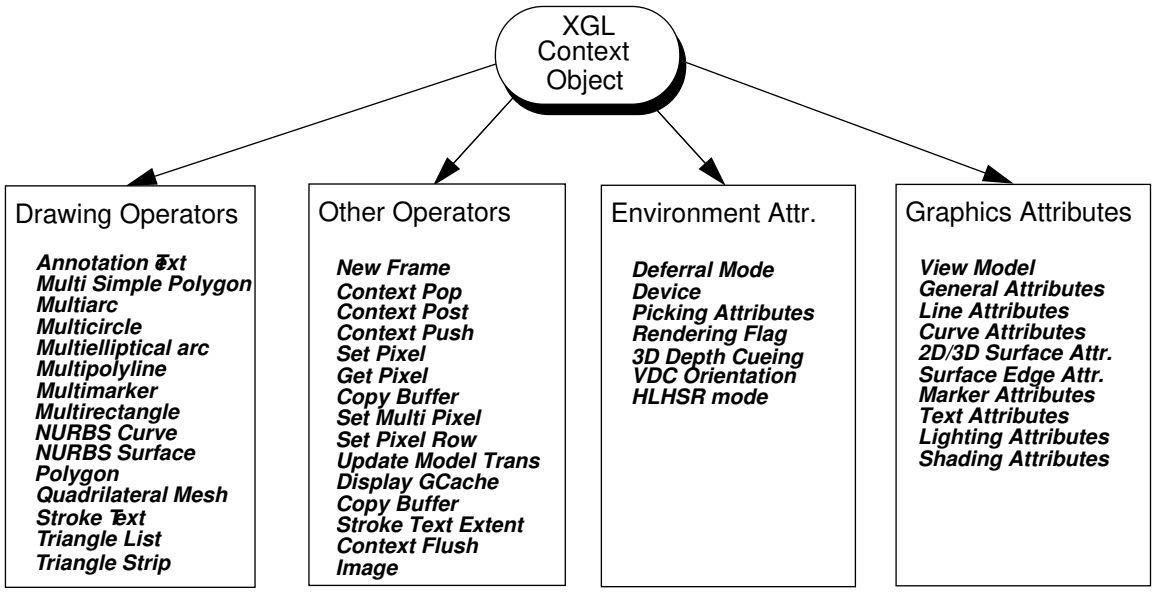

Figure 1-6 Context Object Operators and Attributes

Color Map Object

The Color Map object is an abstraction of a color table. The Color Map specifies colors in a device-independent manner and allows the XGL application to work with its preferred color model. When attached to the Device object, the Color Map object ensures optimal correspondence between the application's color model and the color model of the device.

The XGL library supports indexed and RGB color models. The Color Map object defines color value conversions from one color type to another. If the underlying hardware is indexed, the application can request RGB colors, and the Color Map object performs the RGB-to-indexed color conversion using a color cube. If the underlying hardware is RGB, the application can request indexed colors, and the Color Map object converts the color values using a color table. Figure 1-7 illustrates the two color models. Chapter 6, "Color" on page 121 provides more detailed information on the Color Map object.

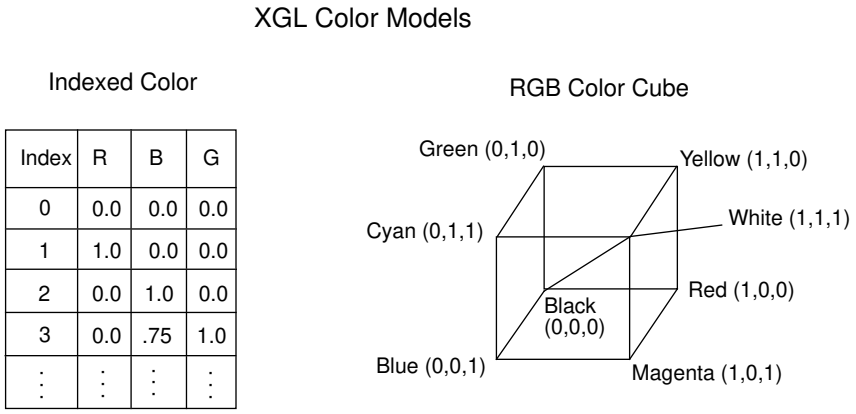

Figure 1-7 Overview of the XGL Color Models

Transform Object

The Transform object provides attributes and operators for geometric transformations performed on graphics primitives. The Transform object represents a set of functions for geometric transformation operations as well as a set of transformation matrices used to map geometry from one coordinate system to another on its way through the rendering pipeline. A single 2D

Context or 3D Context object may use several Transform objects, which in turn can be shared among several Context objects. Figure 1-8 shows geometry that has been rotated and translated in 2D space using Transform object operators. Transform objects are discussed in Chapter 10, “Transforms” on page 273.

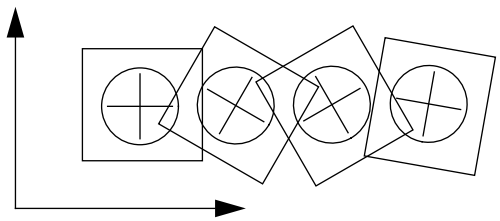

Figure 1-8 Transformed Geometry

Line Pattern Object

The Line Pattern object allows an application to specify a pattern for lines or for surface edges. The application can explicitly define the line pattern, or it can use any of the line patterns that the XGL System State object defines internally. Line Pattern objects can be shared between Context objects and may be referenced simultaneously by more than one Context object. Figure 1-9 illustrates line patterns. See Chapter 13, “Line Patterns” on page 363 for more discussion of the Line Pattern object.

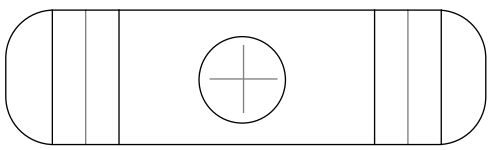

Figure 1-9 Line Pattern Example

Light Object

The Light object allows a 3D Context object to render graphics 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 the application with flexibility, permitting individual Contexts to share some lighting attributes (such as position) but not others (such as light types enabled).

Figure 1-10 shows the four available types of light sources. Chapter 16, "Lighting, Shading, and Depth Cueing" on page 403 discusses the Light object and the XGL illumination model.

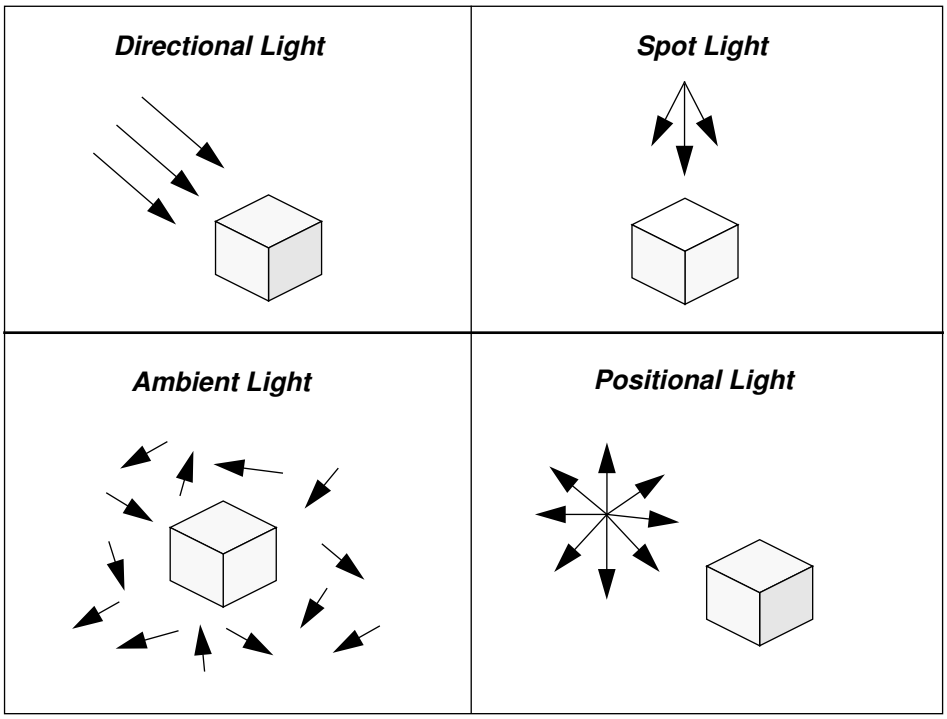

Figure 1-10 Overview of Light Sources

Stroke Font Object

The Stroke Font object allows an application to render stroke text in a variety of font types. By default, the Context object renders a monospaced Roman font. An application can use additional fonts by creating Stroke Font objects and associating them with the Context object. Stroke Font objects can be shared between Contexts and can be referenced simultaneously. Text can be rendered in any orientation in 3D space using the stroke text primitive, or it can be rendered parallel to the display surface using the annotation text primitive. Information on XGL stroke and annotation fonts is provided in Chapter 12, "Text" on page 319.

Marker Object

The Marker object allows applications to describe their own markers or symbols and share them between different Contexts. An application can define its own marker or use any of the predefined markers that the XGL library provides. A user-defined Marker can contain as many as 128 vectors. Figure 1-11 shows a spline with markers designating its control points. See Chapter 14, "Markers" on page 381 for information on the Marker object.

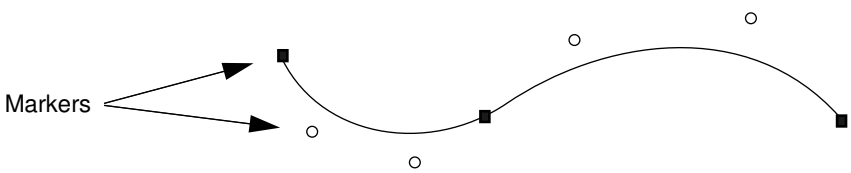

Figure 1-11 Marker Examples

Pcache Object

The Pcache object provides non-editable, non-hierarchical display list functionality in 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. For more information on Pcaches, see Chapter 17, "Caching Geometry" on page 437.

Gcache Object

Gcache objects (Geometry Cache) reduce the complexity of application geometry by processing the geometry into simpler forms. This object reduces complex primitives into many simple primitives, storing the relevant attribute values. For more information on Gcaches, see Chapter 17, “Caching Geometry” on page 437.

Texture Map Object and MipMap Texture Object

The Texture Map object and the MipMap Texture object enable applications to texture 3D surface primitives using a texturing raster image specified by the MipMap Texture 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), as illustrated in Figure 1-12.

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 value and supports several color composition techniques, such as blend, decal and modulate. See Chapter 18, “Texture Mapping” on page 457 for more information on texture mapping.

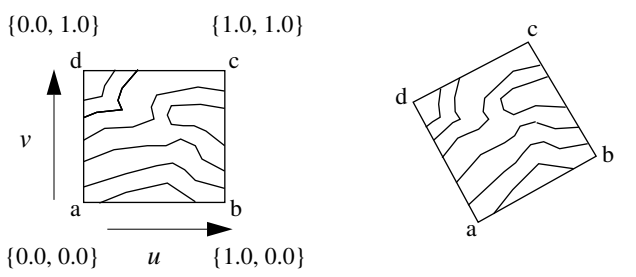

Figure 1-12 Texture Mapping of a MipMap to a Polygon