

XGL: A Case Study of the Design of a Graphics Interface

*Richard Hagy, Patrick Maillot
Sun Microsystems, Inc.
2550 Garcia Avenue,
Mountain View, CA 94043*

Abstract:

This paper presents the XGL API. XGL is an immediate-mode graphics library which provides 2D and 3D graphics in the X environment. An overview of the XGL object-based model is presented, and details are given on the objects, primitives and attributes available to the application programmer. A study of the API is made, and its relationship to current graphics standards is provided in an analysis of its design. The integration with the X window systems is described with its features and drawbacks. An implementation overview is then presented, with a study of some limitations encountered during the development of XGL.

XGL[†]: A Case Study of the Design of a Graphics Interface

*Richard Hagy, Patrick Maillot
Sun Microsystems, Inc.
2550 Garcia Avenue,
Mountain View, CA 94043*

Introduction

XGL is a 2D and 3D immediate-mode graphics library which provides high-performance graphics in the X Window System. The XGL API (Application Programming Interface) provides an interesting case-study of the design decisions taken to create a graphics library interface which supports the needs of a wide range of applications, exploits the capabilities of graphics accelerator hardware in the X window system, and uses the vocabulary and models of the ISO graphics standards. This paper begins with an overview of the XGL library, followed by an analysis of the design goals of XGL and how they affected it. Finally, we review some issues involved in implementing XGL.

XGL Overview

The XGL imaging model is based on that of GKS [ISO85]: there are primitives, which specify the interpretation of geometry, and attributes which control the visual appearance of the primitives. However, XGL diverges from GKS in the model the application uses to program a graphics system. Instead of using an abstract graphics device as GKS does with its “workstation” concept, XGL presents the programmer with a set of “objects” which compose a graphics system; the programmer can use the objects how they best fit his or her application. The set of XGL objects includes Transforms, Contexts, Devices and Color Maps. They are explained in detail below. However, we must first define the term “object”.

XGL objects are a collection of private state, external state and operators. The private state is not visible to the application programmer. The external state is a collection of named values which are visible to the programmer. An attribute is the name of a single piece of external state; the programmer may change the value of the attribute by invoking the operators `xgl_object_set` and `xgl_object_get`; changing the external state may indirectly affect the private state. Each object has a set of operators which can modify the state of the object (e.g. multiplying two transformations or drawing a line in a window).

In “object-oriented” parlance, the external state values are instance variables of the objects and the operators are the methods. However, XGL is not an “object-oriented” system because it does not support classes. The application cannot define new classes (and thus, there is no inheritance). XGL was designed to be a graphics library whose primary interface is in the C language; it is not an object-oriented programming environment. Thus, we feel that it is best to refer to it as an “object-based” system [Weg90].

XGL Objects

The two primary objects of XGL are the Device object and the Context object. XGL provides the application programmer with two flavors of Context object: a 2D Context, and a 3D Context. We explain further in this paper the justifications for such a separation. XGL also provides a Transform object, a Light object, a Stroke Font object, a Line Pattern object, a Marker object, a GCache object, and a Color Map object. In the following paragraphs, more detail is given on the purpose of each object.

[†]XGL is a trademark of Sun Microsystems, Incorporated.

- *Device object*: an abstraction of a drawing surface. This typically represents an X11 window of a workstation called Window Raster, but can also be a memory array called Memory Raster, or a CGM file. The Device object can be attached to one or more Context objects for them to perform drawing operations.
- *Context object*: a Context object can be 2D or 3D and contains the state information which controls the position and appearance of the primitives. A primitive is an operator of a Context whose operand is geometrical information and whose result is the rendering of the geometry.
- *Transform object*: an abstraction of a transformation matrix. It represents a class of objects used by XGL as well as a set of utility functions for the user. Transforms are used in all geometrical computations from one coordinate system to another and for scaling and positioning graphical entities.
- *Light object*: an abstraction of a light with ambient, directional, positional and spot capabilities. Because lights are objects, they can be shared by several Contexts. Lights can be switched on or off on a per Context basis, and the type of shading is Context-dependent: (a given Light can generate specular reflections when used by one Context, and only diffuse with another Context). This offers the user great flexibility when generating pictures.
- *Color Map object*: an abstraction of a color table. An XGL user can work in his own color model even though the hardware uses a different model. The Color Map object attached to the Device will ensure the best possible correspondence. For example, the user can describe his data using true color components, even when displaying on an index color only display. In this case, the Color Map object will use dithering to simulate true color rendering. Generally, in simple cases, the Color Map object is used as the software interface between the user and the Device's color lookup table.
- *GCache object*: allows the application to request that XGL stores complex geometry in a simpler, cached, form. For example, a stroke text string is cached as multiple polylines.
- *Stroke Font object*: XGL provides the application with a full set of text primitives and different text encoding schemes to allow easy internationalization of applications. Stroke fonts can be shared between several contexts, and contain the complete description of a particular set of characters. XGL 2.0 comes with a tool that allow XGL users to develop or modify Stroke Fonts.
- *Line Pattern Object*: an abstraction of a line pattern. Three of them are available by default with any XGL Context. This object allows XGL users to create new line patterns, and to share them between different Contexts.
- *Marker object*: As in the case of the Line Pattern, XGL provides five default markers. An application may need more marker symbols, and this object is used to create new symbols using line segments.

Primitives and attributes

XGL offers to the application programmer a wide range of primitives. Table 1 shows some of the XGL primitives with a succinct explanation of their effect. The XGL primitives are accessible to the application via a Context object; a 2D context and a 3D context may not offer the same set of primitives. At execution time, each primitive gets from the Context the set of attributes it needs in order to perform its rendering. In some cases, the attributes are part of another object attached to the context (e.g. the color of a light when rendering 3D surfaces). Figure 1 shows the set of attributes that are involved in a rendering operation, depending on the primitive.

<code>xgl_object_create()</code>	Creates an object.
<code>xgl_context_copy_raster()</code>	Copies a block of pixels from one XGL Raster to another.
<code>xgl_context_new_frame()</code>	Clears the DC viewport.
<code>xgl_multimarker()</code>	Draws a list of markers.
<code>xgl_stroke_text()</code>	Renders stroke text.
<code>xgl_multipolyline()</code>	Draws a set of disconnected polylines.
<code>xgl_nu_bspline_curve()</code>	Draws a non-uniform, B-spline curve.
<code>xgl_multi_arc()</code>	Draws a list of arcs.

xgl_multi_simple_polygon()	Draws simple polygons from a list of data.
xgl_triangle_strip()	Draws a triangle strip.
xgl_transform_copy()	Makes a copy of a transform.
xgl_transform_point_list()	Transforms a list of points.
xgl_transform_scale()	Combines a transform with a scale transform.

Table 1: Some XGL primitives.

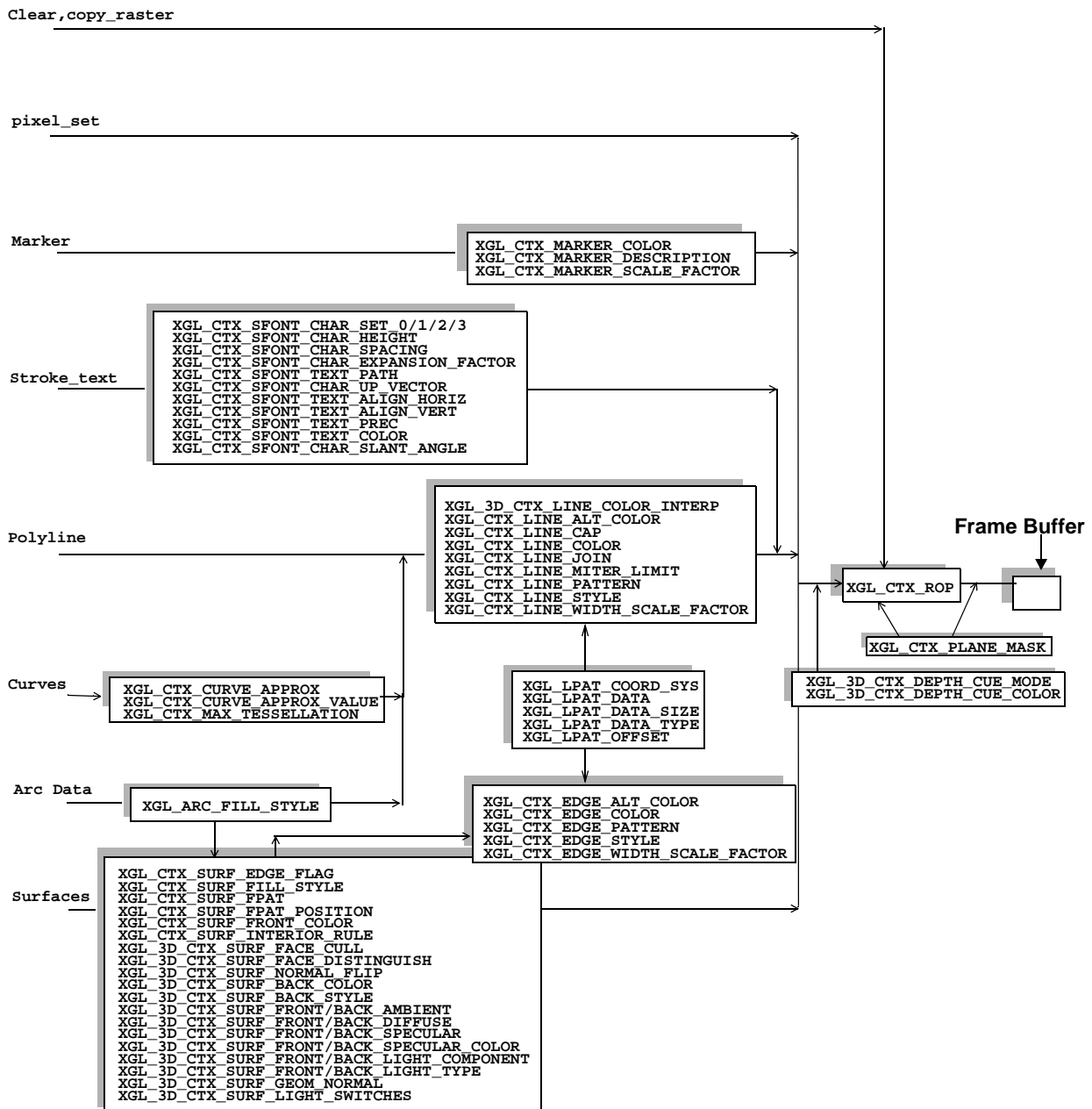


Figure 1: Graphic attributes.

XGL also provides two separate viewing pipelines. A viewing pipeline is a series of transformations and clipping operations applied to a graphics primitive before rendering. XGL's viewing pipeline moves the geometric data from an application's coordinate space to the rendering coordinate space via a set of transformation and clip attributes. The XGL 2D viewing model is an affine model. The XGL 3D view model is a homogeneous model.

The XGL viewing model consists of four right handed coordinates systems: Modeling Coordinates (MC), World Coordinates (WC), Virtual Device Coordinates (VDC), and Device Coordinates (DC). Transform objects are used to map the geometry between any two sequential coordinate systems in the pipeline. Each Context has its own set of transforms for implementing these mappings.

Clipping is conceptually performed in two places: optional clipping may take place in VDC coordinates, and clipping to DC limits is always applied to ensure the correctness of the drawings in an X window. Figures 2 and 3 present the 2D viewing model and the 3D viewing model, respectively.

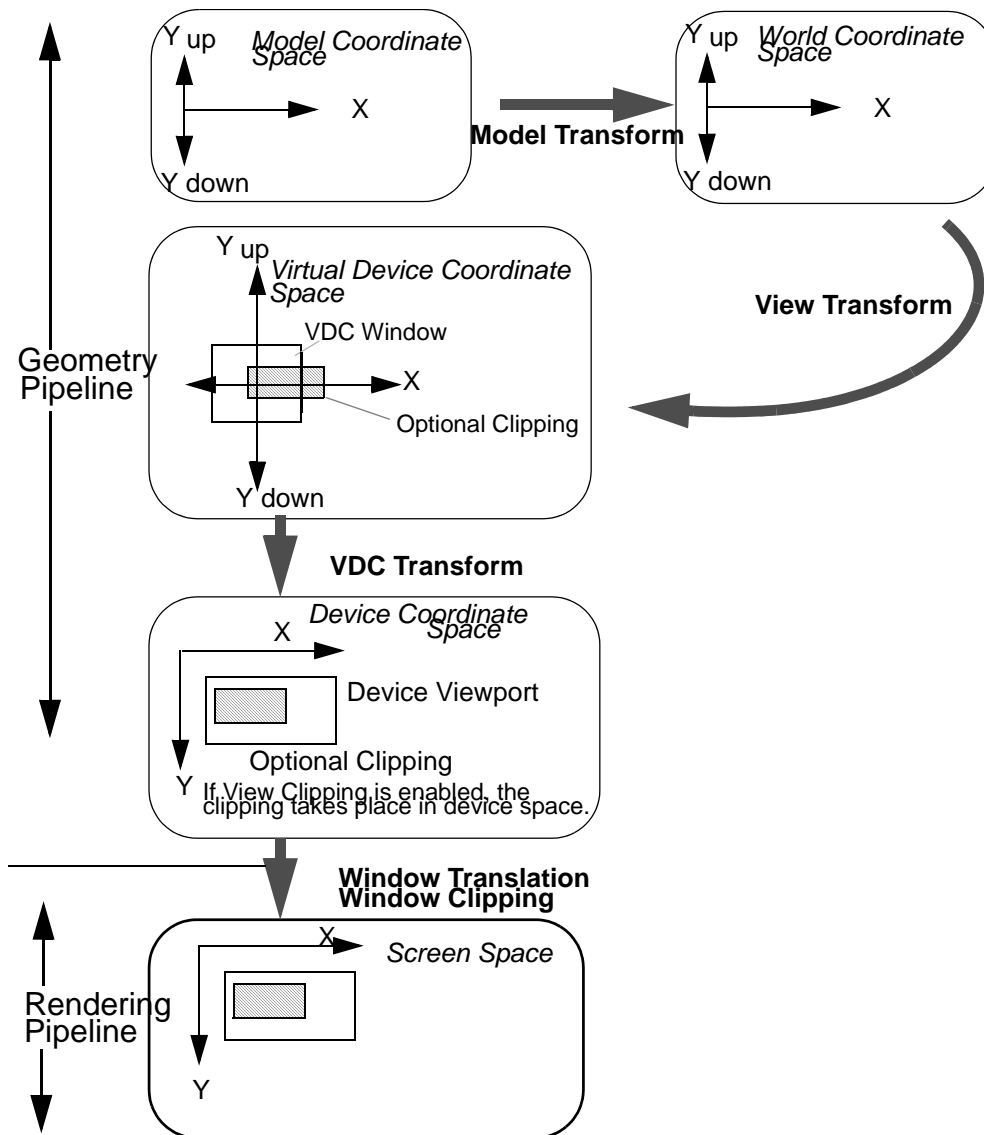


Figure 2: The 2D viewing model.

A unique aspect of the XGL viewing model is that orientation of VDC can be changed. Programmers coming from a raster graphics background are used to having the Y axis going from the top of the screen to the bottom, while other programmers expect the Y axis to go from the bottom of the screen to the top. The XGL programmer can select from either VDC orientation.

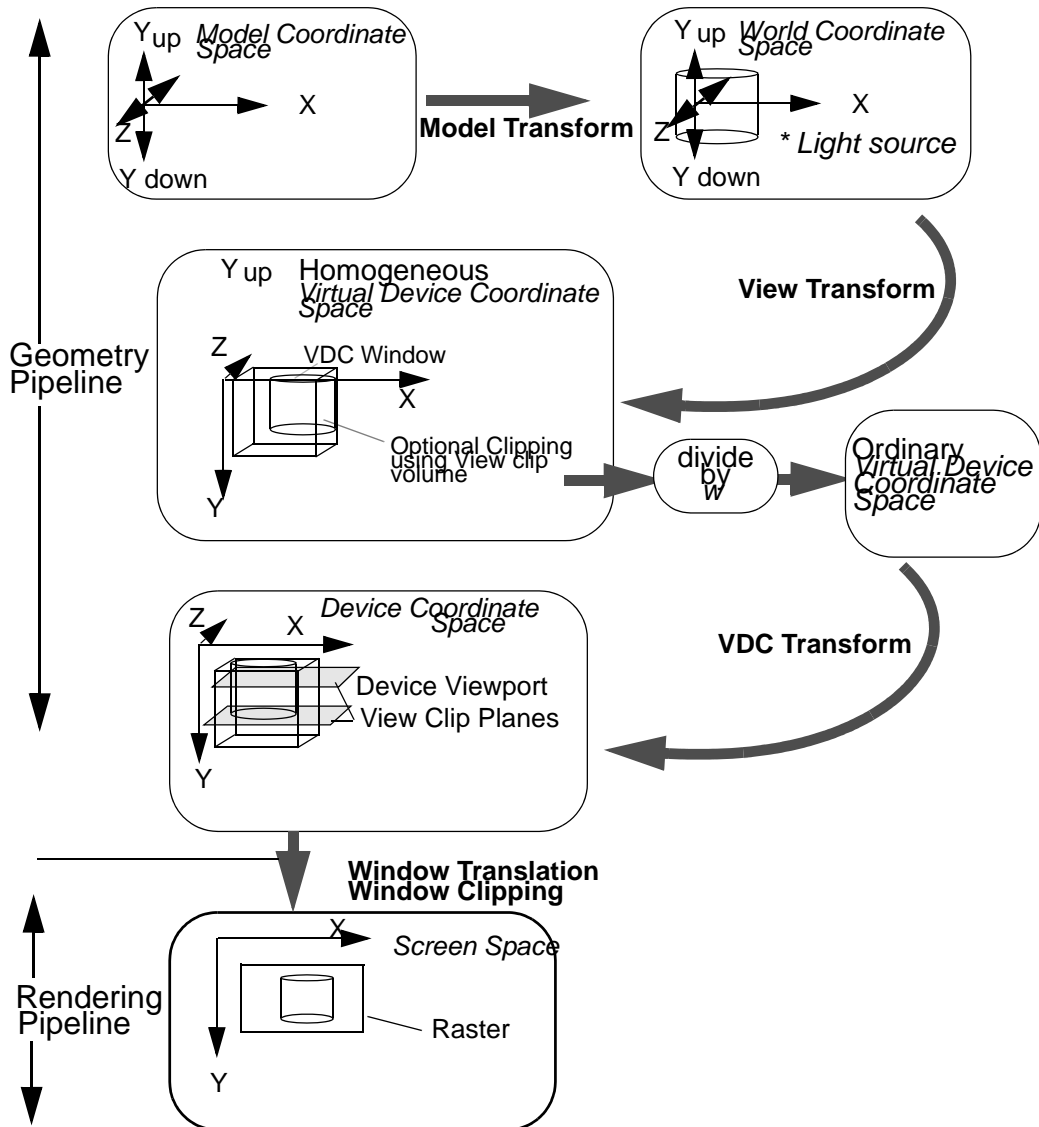


Figure 3: The 3D viewing model.

Analysis of the Design

Previous work

As is evident from the description of XGL, it is an evolutionary graphics API. It is influenced by a number graphics APIs that have come before it. We owe much to the developers of GKS, PHIGS [ISO89], PHIGS+ [ISO90], Sun-View/SunWindows [Sun86], and the X protocol [Sch88]. However, XGL is a new interface and it does reflect a number of design choice that are worth exploring. In this section, we outline the some major design goals and show how they were met in the definition of the XGL API.

Meet Application needs

XGL was designed to support a wide range of applications. The target application domains for XGL includes 2D ECAD, 3D MCAD and implementations of GKS and PHIGS+. The major result of this diverse set of requirements is that XGL treats 2D and 3D primitives in separate pipelines. XGL 2D Contexts are distinct from 3D Context (2D is not a subset of 3D) and therefore, 2D applications can take full advantage of the simpler 2D pipeline. There are two benefits of this decision: performance and space. 2D applications don't require the use of lighting and shading effects at rendering time, and don't need hidden line or hidden surface removed. Also, geometry algorithms such as transformations and clipping can be optimized when applied to specific 2D cases. All of the 2D attributes and most of the 2D primitives are available in 3D, so it is straightforward for programmers to switch between using 2D and 3D.

Another aspect of the XGL API is that in order to satisfy the target application domains, XGL supports integer, single-precision floating point, and fixed-point data types for 2D, and integer and single-precision floating point data types for 3D. Many existing applications, especially 2D ones, are based upon a specific data type. Rather than make these applications be re-written to a single data type, XGL chose to provide them with a choice of common data types.

Integration with the Window system

Because we felt that the primary display device for XGL would be an X11 window, we included integration with the X window system as a major goal of the design of XGL. This goal affected the design of XGL in a number of ways. First, the viewing pipeline had to be aware that the Device (as X11 window) could change size asynchronously. The application must inform XGL that the window has changed size. This ensures that both the application and XGL are aware of the current size of the X11 window and know when it has changed size. (Note: There can be a latency between when the operator adjusts the window size and when the event is reported to the application. During this time, XGL may transform primitives according to the old size. In the case, the primitive may appear in the wrong position in the newly resized window, but the X server will ensure that no primitive is drawn outside the bounds of the window. In practice, this problem does not occur frequently.)

XGL provides the `XGL_CTX_VDC_MAP` attribute which allows the application to let XGL worry about the window resize problem. The `XGL_CTX_VDC_MAP` attribute controls how the final transform from VDC to DC is calculated. Through this attribute, the application can have complete control over this transform by setting the both the VDC Window and DC Viewport, or it can let XGL calculate the DC Viewport. In the latter case, the application does not have to worry about the size of the X11 window and where the DC Viewport is within it. XGL computes a Viewport that either maps the VDC window to all of the Device space (i.e. the full size of the X11 window) or to just the portion of the Device space with corresponds to the aspect-ratio of the VDC Window. A nice side effect of having XGL compute the DC window is that the application becomes more device independent as it does not have to be concerned with specifics of the device's coordinate space.

Another aspect of integration with the window system is the communication and synchronization between the application, XGL and the X server. Since we could only control the XGL portion of the problem, we chose to make XGL as simple and low-level as possible. Thus, since we would not know how the event stream returned from the server would be processed by the application, or by a toolkit, we decided to make XGL output-only. All input prompting, echoing and processing is the responsibility of the application and, perhaps, a toolkit. XGL can be used to provide feed-back for input operations, but XGL does not have the notion of a virtual input device as GKS or PHIGS do. Also, because we did not want to limit applications to using a toolkit, we chose to let the application indicate the window it wants to draw into at the basic Xlib level (a display, a screen and a window) rather than using a toolkit abstraction of a window.

We also recognized that Xlib provides a good model for doing 2D window-coordinate graphics and that XGL could not do everything. Therefore, we allowed the application to mix XGL and Xlib in the same X11 window. However, because of the delay between the time graphics data is sent to the hardware and the time it is actually rendered, XGL requires that the application synchronize with either XGL or Xlib (via, `xgl_context_post` or `XFlush`, respectively) before switching between the two interfaces.

Use Standards when appropriate

The ISO graphics standards provide excellent interfaces for many applications. However, some applications are not serviced by them; for example, not all 3D applications want the display list of PHIGS+. XGL addresses some of the

needs not covered by the standards. However, when designing XGL, we recognized the importance the standards have contributed to the graphics community by defining a vocabulary and reference models for graphics. We, therefore, chose to follow the standard vocabulary and models where it did not conflict with our other goals. This is most evident in the viewing pipeline and primitive-attribute model of XGL. We feel a GKS or PHIGS+ programmer could write an XGL program with very little training in XGL.

There were some areas of the standards which we felt did not fit well with the goals of XGL. Bundle Tables are one aspect of the standards which are not in XGL. We felt that the advantages of not having bundle, a simpler programming interface, and a less complex interface to implement, outweighed any additional functionality gained by having them. Also, we felt that those applications which would want them, could implement them on top of XGL with no degradation in performance.

The graphics standards model of an abstract graphics device is the “workstation”. The workstation has both device independent information (such as the attributes supported) and device dependent information (such as the size of various tables). XGL, on the other hand, has essentially split the “workstation” model into two objects. The Context contains the device independent state information and the Device contains the device dependent state information. For example, a CGM Device has state information concerning a CGM metafile like the description field, while a Window Raster Device has state information concerning the hardware double buffering. A Context can render onto either device, though. We felt that this model for representing an abstract graphical device and graphics state information the most consistent approach for XGL given the decisions to make XGL output-only and not to use bundle tables.

Implementation overview

It was a goal, as well as a challenge, to ensure that all types of devices could be supported by the XGL API, and provide a maximum of acceleration when possible. The first implementation of XGL (XGL 1.0) had to support simple frame buffers, memory devices, X, and the GX accelerator [Mal89]. We decided that the implementation for memory devices would serve as a reference port. It would specify what should be obtained graphically as well as define a common set of algorithms and functions that could be shared by several devices. This turned into an unique architecture design [Mai90].

XGL provides the application programmer with a rich set of 2D and 3D primitives, and a set of attributes that permits control of the graphical output. The software architecture of XGL had to allow for acceleration of this set of primitives on a large variety of hardware platforms. Figures 4 and 5 show the organization of an implementation of XGL for non-accelerated and accelerated devices, respectively.

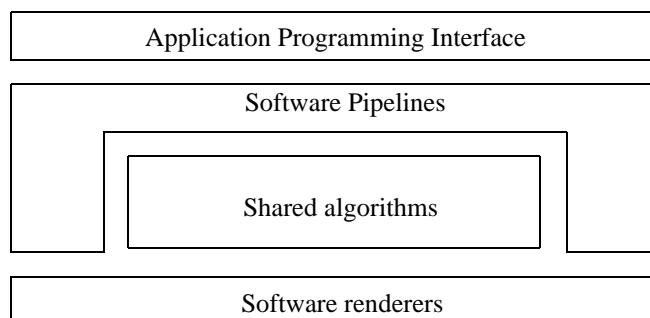


Figure 4: Non-accelerated implementation.

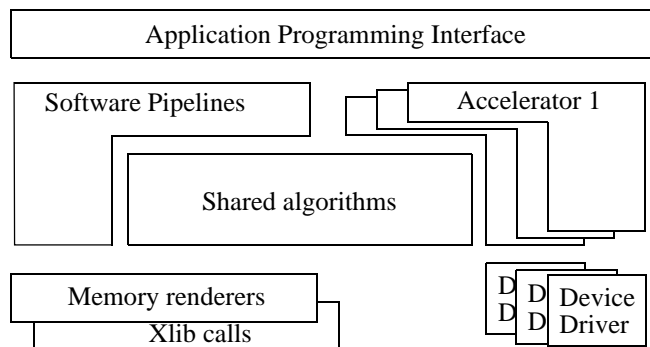


Figure 5: Accelerated implementation.

The XGL API is common to all platforms. In some cases, a specific hardware may be able to accelerate the complete pipeline for a given set of primitives. In that case, the software implementation of the same set of primitives is not used at all, guaranteeing the best performance possible. The X implementation, as presented in Figure 5, involves the complete software pipeline, because XGL provides a complete and comprehensive geometry pipeline including transforms, clipping, and lighting and hidden surface removal in 3D. In all cases, the software implementation can be used as a guideline of what graphics output should be obtained across all devices for a given combination of attributes.

A PEX [PEX89] implementation (XGL emitting PEX protocol) is currently planned for version 3 of XGL and is very comparable to an accelerator port as presented in Figure 5, because PEX provides a high level interface. Some of the primitives of XGL require very little, if any support in software and therefore are as fast as possible for a given PEX implementation. Other primitives, not supported by PEX, or some combinations of attributes outside of the current possibilities of PEX will share some of, or all the software pipeline and the Xlib implementation to ensure correct rendering.

Limitations and imperfections of the model

During the implementation of XGL, we found that we could not guarantee complete device independence in the interface. For example, double buffering would be too slow in a software implementation to be useful. Therefore, some operations are device dependent, and the API allows for inquiry for such features in order for an application to know what is available on a specific device. We kept the set of device dependent features very small in XGL.

Another encountered problem is due to pixel imperfection between two given platforms. The X consortium provided with a standard on what pixel is touched or modified when drawing 2D lines and polygons [Sch88]. Unfortunately, this does not cover for 3D primitives, or differences due to different floating point arithmetic of different platforms (X is integer only, with no transform). Some hardware platforms may even choose other strategies for 2D than the one recommended by the X consortium for different reasons. As a result, it is impossible for XGL to guarantee that device A will draw exactly the same line than device B, even under the same set of parameters and attributes.

Conclusion

This paper has presented some design decisions that have been made during the XGL development process. XGL is an evolutionary graphics API that combines the models of the graphics standards with current object-based programming technics. We feel XGL provides a comprehensive interface that meets the needs of most of today's graphics applications. XGL has proven to be implementable on a wide variety of devices and delivers the performance the hardware is capable of. The XGL interface is positionned to adapt to new graphics functionality and hardware as they become available.

References

- ISO85 ISO 7942 : 1985, *Information processing systems - Computer Graphics - Graphical Kernel System (GKS) Functional Description*
- ISO89 ISO 9592-1 : 1989, *Information processing systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) - Functional Description*
- ISO90 draft proposed 9592-4 : 1990, *Information processing systems - Computer Graphics - Programmer's Hierarchical Interactive Graphics System (PHIGS) - Plus Lumi'ere und Surfaces (PHIGS PLUS)*
- Mai90 P. G. Maillot, "XGL Architecture," *Suntech Journal*, May/June 1990, pp 10-22.
- Mal89 C. A. Malachowsky, C. R. Priem, "The GX Graphics Accelerator," *Suntech Journal*, Autumn 1989, pp 85-93.
- PEX89 PEX Architecture Team, *PEX Protocol Specification Ver. 3.30*, M.I.T., 1989.
- Sch88 R. W. Sheifler, J. Gettys, R. Newman, *X Window System, C library and Protocol Reference*, Digital Press, 1988.
- Sun86 Sun Microsystems Inc., *SunView^(tm) Programmer's Guide*, Sun Microsystems Inc., Mountain View, CA, 1986
- Sun89a Sun Microsystems Inc., *XGL^(tm) 1.0 Reference Manual*, Sun Microsystems Inc., Mountain View, CA, 1986
- Sun89b Sun Microsystems Inc., *XGL^(tm) 1.0 Programmer's Guide*, Sun Microsystems Inc., Mountain View, CA, 1986
- Weg90 Peter Wegner, "Concepts and Paradigms of Object-Oriented Programming," *OOPS Messenger*, Vol 1, Number 1, Aug 1990, pg 8 and pg 26.