CHAPTER 1.

Introduction

The current state-of-the-art computer operating environment usually provides the user with a high performance workstation having 3D graphics capability. These workstations are typically connected by some high speed network link. Through the network, the users gain access to powerful super or super-mini computers. The users may then run computationally intensive applications on these powerful computers and examine the results on the local workstations. Since most of the applications are complex, examining the results of the applications usually involves some graphical display operations [26]. Other users may use the workstations to perform tasks such as computer-aided design and computer-aided engineering. A significant portion of these applications involve 3D graphics.

There are a wide variety of workstation architectures and graphic rendering engines available. PHIGS [15] is an ANSI 3D graphics standard designed to address the issues of portability and consistency for 3D graphics applications. In order to provide a friendly and efficient working environment, the high performance workstations normally support windowing systems. Currently, the most widely accepted windowing system is the X Window System [3]. PEX [7,8] is an attempt to integrate PHIGS and PHIGS PLUS[1] into the X environment. The advantage of this integration is that the resulting system will be able to support standard 3D applications in a widely accepted windowing environment. UIPEX is a PEX implementation developed by the Graphics Workstation Research Group at the University of Illinois Department of Computer Science.

The main objectives of this thesis are:

(1) To present a design level discussion on the X Window System and its extension mechanisms. The focus of the discussion is on the extension system architecture, extension functional blocks, and the extension operational model.

(2) To describe the PHIGS/PHIGS PLUS extensions to X (PEX). In particular, the issues involved in the implementation of PEX are discussed in detail.

(3) To describe the PEX implementation created by the Graphics Workstation Research Group at the University of Illinois (UIPEX). The advantages and limitations of UIPEX are also discussed.

The X Window System is the most widely accepted windowing system. It is designed to efficiently support 2D windowing operations in a networked environment of heterogeneous workstations. X provides a simple and efficient framework for the extension of the system for the support of special applications. For example, extensions may provide support for PostScript and 3D graphics applications. This thesis first presents a system level description of the approaches to building an

[1] PHIGS PLUS is a proposed ANSI standard.

extension to X. The author is inspired by the lack of publications describing the approaches and design philosophies of the X Window System extension mechanism. The only existing guides to the extension implementation are low level manuals. While these manuals are essential for implementation, they fail to explain the operational model of the extension mechanism and there is no explicit extension architecture overview. The second chapter describes the X Window System through an overview discussion of the issues related to building an extension. Chapter 3 presents the operational model of the X extension mechanism and the author's view of the philosophy and approaches to be used in designing an extension to the basic system.

The second part of this thesis focuses on the UIPEX implementation. PEX is an attempt to extend the X Window System to support PHIGS and PHIGS PLUS. PHIGS and PHIGS PLUS are the ANSI standard for 3D computer graphics. They are defined to support general purpose 3D computer graphics applications with light source shading and primitive reflection attributes. This thesis uses the results of our study of PEX and the solutions used in the UIPEX implementation as a means of carrying out a discussion of the problems encountered in integrating PHIGS/PHIGS PLUS into the X Window System.

Basically, PEX must address three major issues:

(1) Adding the PHIGS semantics to the X Window System. Being a windowing system, X emphasizes resource sharing and user control of the applications. The PHIGS definition has no notion of resource sharing. To facilitate the development of well defined applications, PHIGS defines the application to be in control. PEX must address the problems in combining the two systems.

(2) Storing of 3D modeling information in a distributed environment. Since X is a distributed windowing system, the trade-off between storage of modeling information in the client or in the server is a major design decision.

(3) Extending a 2D system to support 3D applications. X provides the support for standard 2D graphics applications [6]. When integrating a 3D system into X, one design decision involves the means by which the 3D system can take advantage of the more efficient 2D facilities.

In order to understand PEX and the issues involved in the PEX implementation, the design philosophies and the system operational models of PHIGS and the X Window System must be studied. Chapter 4 provides the necessary PHIGS and PHIGS PLUS background for the PEX discussion. Chapter 5 presents the PEX architecture and operational model. Chapter 6 discusses implementation issues that are not addressed by the PEX definition. The implementation details of UIPEX are presented in chapter 7. The thesis concludes with a discussion of the problems with the PHIGS and PEX definitions.

CHAPTER 2.

The X Window System

The *X Window System* (X) was developed at the Massachusetts Institute of Technology (MIT), as part of project Athena. The main purpose of X is to provide a windowing environment for the development of network transparent[1] and device independent applications. Since the Athena project environment included a variety of hardware architectures, portability of the implementation was another goal [3]. The first widely accepted version of X was the tenth (X10). Since that time the network protocol has been revised and the current widely accepted X version is the eleven-th (X11). At the time of this writing the latest release of X from MIT is version eleven release three (X11R3)[2].

This chapter presents the basic model of X. The focus of the discussion will be on the system dependent information that affects the implementation of an extension. The design philosophy of X will also be discussed.

2.1. X System Model

The origins of X lie in the Stanford University W window system [3]. As with the W system, X allows network transparent access to a hierarchy of resizable and overlapping windows. X improves on the W window system by introducing an asynchronous network protocol replacing the synchronous W protocol. As reflected by figure 1, the basic model of X consists of a pair of cooperating client (GKS App, xterm, Window Manager) and server (X server) programs. X itself is defined by the network communication protocol between the cooperating clients and servers. In other words, X defines what message the client can send to the server and vice versa. The operational model is implied by the communication protocol.

The client applications are device independent[3]. They send requests to the device dependent servers to accomplish their tasks. Multiple client applications can be connected to a single server and vice versa. The server can be thought of as an abstract hardware interface; although the hardware on

[1] *Network Transparent* means that the application need not be aware of which machine the output is going to be displayed on. An application will perform the same operation to open a window on a local machine or on a remote machine on the network.

[2] The version number indicates a different network protocol, and the release number indicates implementation enhancements. Since there is no intrinsic difference between releases of the same version, from here on *X* will be used to refer to all releases of X11.

[3] *Device independent* is used to refer to the fact that the client application can produce output on any X server independent from the underlying display device. Notice that the consistency of the output may not be preserved across different hardware. For example, since the drawing coordinate is defined to be the display device pixels, the output of the client application may be different on different display devices.

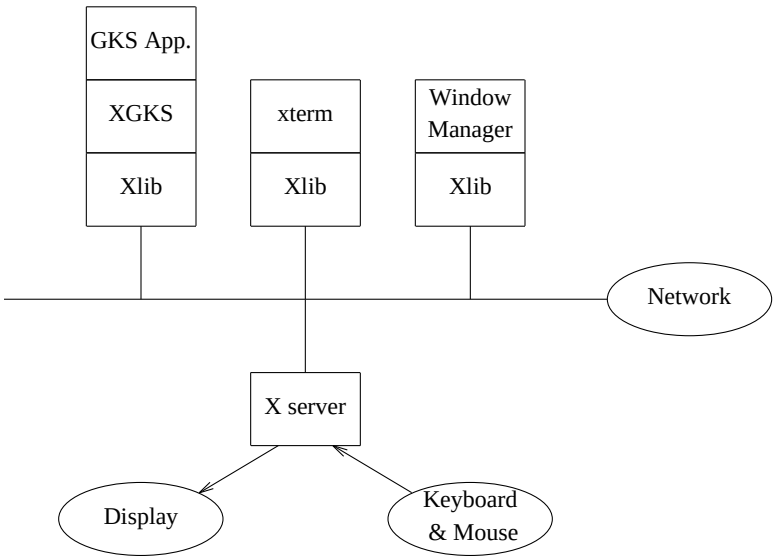

Figure 1: X System Model

which the server operates can be different, the interface (the X protocol) to the client remains the same.

The *Xlib* is the client side X protocol encoder. It is responsible for converting the client requests to a network format and sending the converted requests to the server. In addition, Xlib also caches the client requests[4] and performs simple error checking. The X server receives the packets sent by Xlib and processes them to carry out the client's commands. X defines four types of network packets: *request*, *reply*, *error* and *event*. The client application is the sender of request packets and the receiver of the reply, error, and event packets. The idea is that the client sends requests to the server and receives replies or possible errors from the server. The server sends event packets asynchronously to inform the client about the environmental changes that concern the client application. For example, the server will send an event to the client when the client window is iconized.

A fundamental concept in X is that the server *''serves the client and makes no policy decisions''* [3]. This idea keeps the system simple and efficient; the application decides what should happen and sends the requests to the server to carry out the decisions.

[4] The client requests are cached in Xlib, and sent out in batches to utilize the network efficiently.

2.2. Resources

The operational model of X is that the client creates and manipulates *resources* in the server. The server maintains these resources on behalf of the client. The client coordinates the resources in the server to perform the desired operations. Figure 2 shows the cooperation of the *Graphics Context* (GC), the *Window*, the *Cursor* and the *ColorMap* resources to produce output on the display screen. In this example, the client application first sends requests to the server to create the four resources. After creation, the client may send additional requests to set the resources' states and to associate related resources (e.g. associate the Cursor and ColorMap resources with the Window[5]). After the initial manipulation, the client application can send requests to display graphics primitives on the Window resource. Conceptually, the GC resource processes the output primitive commands and generates an image on the display device through the Window resource. The colors of the picture being displayed are defined by the ColorMap resource.

Although the above example shows only a small subset of all the possible resources in the server, it indicates the basic operational model of the X server. The server maintains resources on behalf of the client, and the client coordinates these resources to perform the desired operations. Although this model is simple and elegant; there are certain issues that need to be discussed:

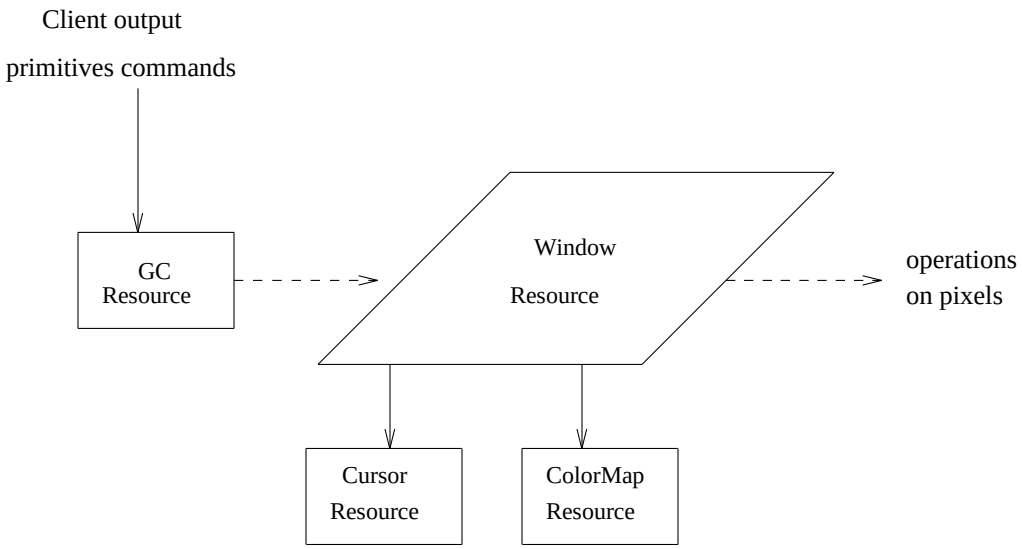

Figure 2: X Resource Example

[5] The Cursor and ColorMap resources support the Window resource's operation. In X terminology this is described as ''the Window resource referring to the Cursor and ColorMap resources''.

Resource Identifiers

One problem with the X model is that the server must maintain resources of the same type for different clients. For example, in figure 1 the GKS application and xterm both have their own Window resources in the same server. How can the server differentiate which Window belongs to which application? In order to resolve this problem a unique *Resource Id* is defined for each resource created by the application. When an application first opens a connection the server assigns a *resourceIdBase* (an unsigned number) to the Xlib. Each time the client creates a new resource Xlib increments the *resourceIdBase* and assigns the resulting number as the new resource's id. Each client is guaranteed to have a large number (on the order of 2^{20}) of unique resource ids. The server associates the resource id with the physical resource storage. This id is the only way for the client to access a resource in the server. Since it is unique, the server is able to locate the specified resource. Under this scheme it is virtually impossible for two clients to have the same resource ids and there can be no accidental resource access.

Reference Count

The client coordinates the resources by creating *resource references*. In figure 2, the Window resource refers to the ColorMap resource for color definition and the Cursor resource for cursor support. A problem that arises is determining what should happen when the client destroys resources currently being referenced (e.g. the ColorMap and Cursor in figure 2). X has two approaches to solving this problem:

(1) A reference count is kept for all resources. The reference count is incremented each time the resource is being referenced (e.g. when the Cursor is being referenced by the Window), and is decremented when the referring resource releases the reference (e.g. If the Window resource is being destroyed before the Cursor resource, the Window's reference on the Cursor will be released). All destroy operations on resources having a reference count greater than zero will be deferred until the reference count becomes zero.

(2) The server defines a default resource for each resource type that cannot be destroyed by the client. For the example of figure 2, the server will obey the client's command to destroy the ColorMap resource and change the Window resource's ColorMap reference to the default system ColorMap. In this case, there may be a noticeable visual distraction. For example, if the default ColorMap and the one being destroyed are different, there may be a sudden color change on the Window resource.

X defines the above two ways to handle the destruction of currently referenced resources. However, there is no mention of which method should be employed in a specific situation. It is not clear why X provides the different solutions. To this author the choice for the solution seems quite arbitrary.

Resource Sharing

Different applications can share resources in the server. For example, two applications may decide to share the same ColorMap resource for their Windows. The only way to achieve this is for the non-owner[6] application to communicate with the owner application to obtain the resource Id. The non-owner application can then manipulate the resource in exactly the same way as the owner application. For example, Application_A has created ColorMap_A. The only way for Application_B (the non-owner) to gain access to ColorMap_A is for it to communicate

[6] The *owner* is the application that has created the target resource.

with Application_A (owner) to obtain the resource id of ColorMap_A. After getting the resource id, Application_B can manipulate ColorMap_A in the same manner as Application_A. One important point is that the X server destroys the resources for a terminating client. In this case Application_B must realize that ColorMap_A will become invalid after Application_A terminates.

2.3. Multiple Display Support

The X server is designed to support multiple display screens simultaneously. For example, it is possible to run an X server on a system with both color and monochrome displays, and provide to the client application access to both of the displays simultaneously (see figure 2). The X server implements this functionality by defining the *Screen* structures to encapsulate the different hardware displays. The Screen structure provides the server with a consistent and abstract interface to the underlying display device.

The Screen structure defines a set of procedure pointers through which the server interacts with the display device. There are procedure pointers for all hardware dependent functions. For example, there are procedures for the creation of Windows because creating a window is a display dependent operation. The GC is the encapsulated 2D rendering engine. Depending on the display device's capability, some of the rendering functionality may be realized by the hardware. When the server process first starts up, it determines what display devices are present. A Screen structure is created for each of the display devices and the procedure pointers in the Screen structure are filled by the interface library of the respective device. Hardware dependent resources (e.g. Windows, GCs) always have a pointer reference to the Screen structure for device specific information. Notice in figure 3 that the server would not allow the Windows created for the color monitor to cooperate with the GC created for the monochrome monitor since their target display devices have different capabilities. Finally, notice that the Screen structure is not an X server resource and it is not accessible by the clients.

2.4. Network Transparency

In order to provide network transparent access across machines with different architectures, the potential differences in byte ordering, floating point format, and color format must be invisible to the application. When sending packets across the network the system must retain the correctness of the information contained in these data items. The X protocol avoids the floating point and color format problems by not defining any packets with such information. The X server is responsible for keeping the byte ordering consistent with the client application. When a client first establishes a connection to the server, the server checks and sets the *ClientByteOrderingFlag* to indicate the client's byte ordering. For each received packet, the server consults the *ClientByteOrderingFlag* and performs byte swapping when necessary.

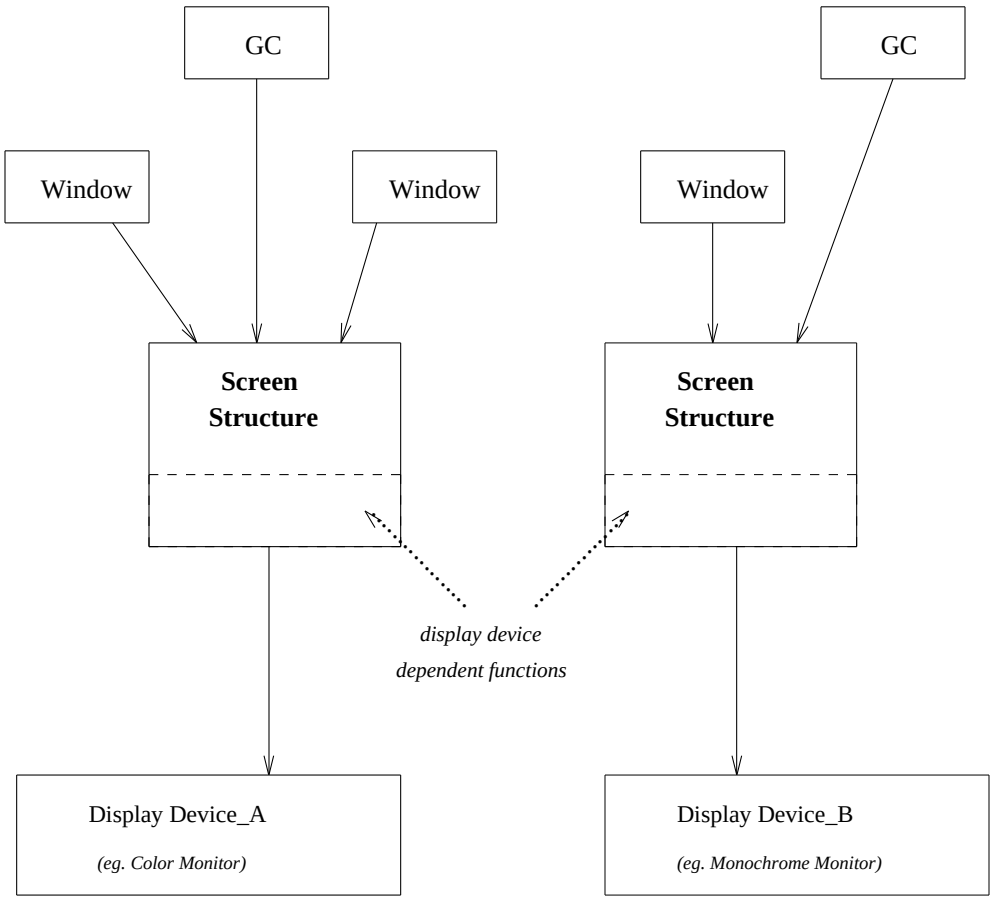

Figure 3: The Screen Structure

2.5. X Design Philosophy

The basic philosophy of X was to design a system that performs 2D graphics and windowing operations efficiently. Since X is a windowing system, resource sharing and resource managing responsibilities are very important issues. The definition of X has carefully taken these into consideration. A set of guidelines on how system resources should be handled are laid down for all applications. The working environment assumed by X is a group of high performance workstations connected by a high speed network that shares display and computing resources. The client and server system model is defined so that the server assumes no responsibility for policy making. In particular, when and what should be drawn on the output display is completely controlled by the client applications. X adopted the *external* control [22] user interface model. A typical X application performs the necessary initialization, and then enters a polling loop where it performs the commands

indicated via user interaction.

CHAPTER 3.

Building An Extension To X

X is designed to support efficient 2D graphics and windowing operations in a distributed environment. It follows a *simple and efficient* design guideline [27] and establishes a well defined extension mechanism for the development of other applications in the X environment. Although not stated explicitly, it is important for the extensions to follow the X design philosophy. In this way the characteristics of X are retained and all applications will work with an extended system, rather than with only a modified system. By following the core X[1] philosophy, the extended system will retain support for, maintain the integrity of, other X window applications. In particular, the following guidelines should be followed by the extension designers:

(1) No single application should assume the ownership of any hardware resource.

(2) The server should never make any decision and should hide all the possible format differences from the client. This means that the possible floating point and color format conversions should be performed in the server and should be completely invisible to the client applications.

(3) The extension should follow X's cooperating resource model.

Neither X nor the extension mechanism enforce any of these ideas. It is entirely up to the application and the extension designers to comply with them. The extended X should be considered to be a system composed of independent functional blocks which cooperate to support the applications. The aim is to keep each functional block simple, efficient, and independent from the other blocks. The major advantage of this approach is that the functional blocks can be refined and upgraded independently.

One problem the extension designer may encounter is overlapping of functionalities between core X and the targeted extension system. This problem involves whether the extension should redefine the existing core X functionalities or whether it should burden the application with coordination of the independent functional blocks. This is a serious issue. Since extensions are defined to be loosely coupled to core X, coordination of the extensions and core X could be difficult. For example, core X only has adequate functionality to support standard 2D applications [6]. Should a 3D extension overlap this functionality by defining an orthogonal set of 2D primitives, or should it exclude 2D primitives and delegate the burden of coordinating the core X 2D primitives and the extension 3D primitives to the applications?

[1] Henceforth *core X* is used to refer to the original X Window System and *core server* is used to refer to the original X server.

3.1. Preview

There are two existing documents on how to extend X. One describes the implementation of the client extension library and the other discusses the procedures for developing an extension server. The client side documentation is in the appendix of the Xlib manual [21]. The extension server documentation was originally distributed with the X Version 11 Release 1, dated August 1987 [20]. Since then, there have been two new releases of X Version 11 but there have been no updates to this document.

This chapter summarizes the above two documents with emphasis on the extension system overview and basic philosophy. The *Client Side Extension Library* and the *Extension Server* are discussed separately with the focus on their responsibilities and operational models. The original manuals [20, 21] are helpful in developing actual implementations. The goal of this chapter to to provide readers with a better understanding of approaches to designing an extension to X, and a better understanding of how the extension cooperates with core X.

3.2. X Extension Model

The extension mechanism defines the expansion of both the client and the server programs. To conform to the core X system model, the extension must provide a client side application interface library. The server needs to be expanded to perform the new operations. Under this model it is clear that the X extension should also be defined by the communication protocol between the client and the server. The extension mechanism does not define new packet types; the request, reply, error, and event packet model remain the same as in the core X.

Figure 4 shows the conceptual model for extending the X widow system. The extension must always be implemented in pairs: the *Client Side Extension Library* and the *Extension Server*. The client side library is responsible for encoding the application requests into the defined protocol packets. The server extension must decode and perform the client side requests. This conforms to the core X model that the client issues commands and the server performs them.

Core X can be used as the communication link for the extension. All the extension packets are processed and dispatched by the core X routines. On the server side core X examines all the request packets and dispatches them to the core X server routines or the corresponding extension routines. On the client side, Xlib must examine all the error and event packets to determine which extension library should receive them. The rest of this chapter describes these interactions in detail.

3.3. The Client Side Extension Library

The major responsibility of this library is to encode the client requests into packets and send the packets to the server. The extension requests are of the same priority as the core X packets and should be handled in the same manner. One way to implement the extension library is to use the existing Xlib internal network routines to communicate with the server. The advantage of this approach is that the client extension library is network independent and can be easily ported. The major disadvantage is that the extension library's communication ability becomes limited by the

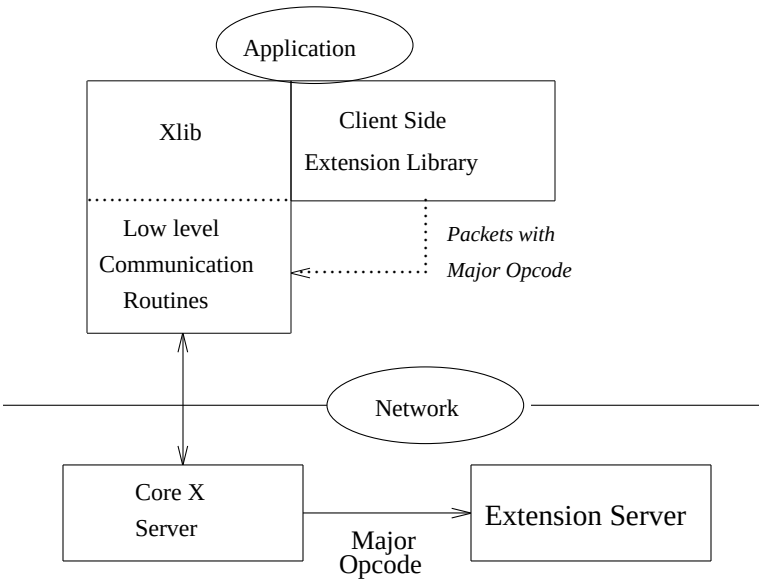

Figure 4: X Extension Model

existing Xlib functionality. Xlib's internal routines are fine tuned to support the core X protocol messages and may not be suitable for the extension. For example, Xlib buffers request packets without considering the message semantics. In some cases it may be desirable to examine the content of the messages to merge the related packets. For example, in cases such as 3D graphics applications, where due to the large amount of modeling information, the packet merging process could reduce the network traffic significantly.

The responsibilities of this library include: extension registration, packet encoding, resource id allocation, error handling, event handling, final system clean up, and graphics context (GC) extension. The rest of this section covers each of these functions in detail.

Extension Registration. A registration function should be provided by the extension library to allow the application to inquire and certify the existence of the server extension. It is the application's responsibility to perform the registration with core X before invoking any of the extension's functionality. The registration process is implemented by calling the Xlib *XInitExtension* function. XInitExtension returns a structure describing the valid error code range, valid event code range, and a unique major opcode[2]. If the extended server does not exist, an invalid structure will be returned.

[2] Each extension is guaranteed to have a unique major opcode. As will be discussed in the *Extension Server* section, the core X server is actually responsible for assigning the unique opcode.

Packet Encoding. All request packets in X (including the extension packets) have identical headers: *MajorOpcode*, *MinorOpcode*, and *PacketLength*. The MajorOpcode field of the packet header is filled with the unique major opcode obtained during the extension registration. An extension is likely to have more than one request type. Defining a different extension for each request type is not acceptable. An extension should encapsulate the desired functionality and define the minor opcode so as to indicate different request types. For example, a 3D extension could define Output3DText and Output3DLine as two different request types with distinct minor opcodes. The PacketLength field would contain the length of the request packet. The client extension library is responsible for packaging the packet header and the extension specific information into the network format and sending the resulting packet across the network. Typically, there are different sets of data for each request type. For this reason, one way to implement a client extension library interface is to define a procedure for each of the defined minor opcodes. For the above 3D extension example, the client library could define *XExtension3DText()* and *XExtension3DLine()* interface functions for the application. These functions will pack the application data into a network packet, fill the request packet header with the extension specific information, and send the packet to the server.

Resource ID Allocation. It is desirable for the extension to follow the core X philosophy by modeling the server as a set of cooperating resources. Xlib provides the *XAllocID()* routine for the extension library to allocate unique resource ids. This allows the application to create and access extension-specific server resources.

Error Handling. During the extension registration, a valid error code range is assigned to the extension. The extension server should send all extension specific error packets with an error number in this range. To support error handling correctly the extension library must register two functions with Xlib. The *XESetError()* registers the error handling routine, which is activated when an error packet with the error number in the corresponding range is received. The *XESetErrorString()* registers the error message output routine. When Xlib receives an error packet that it does not understand it sequences through all the error handling routines registered by the *XESetError()* to find the owner of the packet. The extension's error handling routine should examine the error packet and raise the *StopFlag* if the error number is in the range of the extension's valid error codes. When the StopFlag becomes *True*, Xlib stops the ownership search and the corresponding routine registered by the *XESetErrorString()* is invoked to handle the error (usually, by printing out messages). This mechanism allows the extension to take advantage of the existing core X error handling facilities. At the same time, the extension is allowed to control the erroneous situations.

Event Handling. As with the extension error range, the extension event range is assigned during the extension registration. For each extension defined event, *XESetWireToEvent()* must be called to register a function with Xlib to convert the received event packet from network format to the application specific format. After the format conversion, the extension events are treated exactly the same way as the core X events.

Final Clean-Up. The extension library is responsible for cleaning up its internal data structures when the connection to a server is closed. The *XESetCloseDisplay()* sets the extension shut down routine. When the connection to a server is closed, Xlib invokes the function registered by the XESetCloseDisplay(). This function should destroy all the extension specific data structures.

Graphics Context (GC) Extension. This could be a useful mechanism to extend the core X's 2D output functionality. For example, the GC can be extended to include curve drawing capabilities. Alternatively, the extension designer can define new resources for such a purpose. The advantage of

defining new resources is that the extension can be cleanly separated from core X thus facilitating porting and independent refinement. However, new resource definitions hinder the use of existing GC functionality. For example, by defining a new resource to realize curve drawing capability it is difficult to take advantage of the GC clipping function. In the early stages of the PEX [8] definition all of the 3D functionalities were defined to be extensions of the GC. After refinements, the designers rejected the approach and chose to define new server resources. The practicality of this extension mechanism is not clear; it is the author's opinion that it is an artifact of the prototype extension design.

3.4. The Extension Server

The extension server is loosely coupled to the core X server. The only functional link between the extension server and the core server is that the extension server receives the request packets from the core server. Although not defined, the extension could send reply and event packets through the core server routines. Of course, when necessary the extension server can access the core server's functionality. The extension server should be modeled after the core server as a set of cooperating resources. The core X concept that the server does not make any decisions should also be obeyed by the extension designers. The rest of this section describes the responsibilities of the server extension and its interaction with the core X server.

Core Server Modifications. The extension server must be integrated into the core server by modifying the core server's source code. The *InitExtensions()* function of the core server must be edited to add in the extension initialization routine. When the extended server starts up, all the routines in the InitExtensions() will be invoked to allow the various extensions to initialize themselves.

Extension Server Initialization and Registration. The routine being invoked by the core server's InitExtensions() must perform the extension registration and initialization. The extension registration is performed by calling the core server's *AddExtensions()* routine to achieve the following functionality:

> **Extension Name Registration**. A name unique to the extension should be registered with the core X server. In return the core server will assign a unique major opcode to the extension. Thereafter the extension server will be identified and accessed exclusively by the major opcode.
>
> **Error and Event Code Request**. The extension requests a set of valid error codes from the core server and an error code range is assigned to the extension. The extension server must send its error packet using a code within the valid error code range. The same mechanism is used for extension event codes.
>
> **Dispatcher Registration**. The extension registers a pair of procedure-pointers with the core server. These procedures are the extension dispatchers; one for byte swapped packets (*SwapDispatcher*) and the other for normal (*NoSwapDispatcher*) packets. The core X server will invoke the corresponding dispatcher upon receipt of the extension packets.

The extension initialization typically involves the creation of the resource *type* and *classes*, the making of *X Atoms*, and the initialization of the extension-specific data structures. Resource *type* and *classes* will be discussed in the next section. *X Atoms* are the names of the server's global properties.

The client applications can use these names to access and make inquiries about the corresponding properties. The existence of the extension-specific atoms can be used to indicate to client applications that the extension server is present. During the client side extension registration, Xlib uses the extension's unique name to obtain the corresponding error codes, event codes, and the unique major opcode from the core server.

Resources Handling. It is desirable to preserve the philosophy that the server is a set of cooperating resources. One convenient approach is for the extension server to define and access resources the same manner as is done in the core X server. X categorizes the resources into *ResourceTypes* and *ResourceClasses*[3]. An extension server should define a different resource type for each of the cooperating resources. Resource classes can then be defined to capture similar functionalities of different resource types. For example, the core X defines RT_Window and RT_Pixmap resources types for Window and Pixmap resources. The Drawable resource class, RC_Drawable, is defined to be the set of Window and Pixmap resource types. Under this scheme the core server unifies the handling of the overlapping functionality of the Window and Pixmap resources. After the AddExtensions() function returns successfully, the extension server's initialization routine should create all of its resource types and classes. In this way all of the extension resources can be handled by the core server's resource manager and be treated in exactly the same way as core X resources. As with the core X server, the extension server must bear the responsibility for keeping the reference counts for all of its resources.

Decode and Dispatch Scheme. The core X server examines the MajorOpcode field of each packet it receives and dispatches the packet accordingly. Since the client library has filled the MajorOpcode field with the extension's major opcode, core X can identify the extension packets and invoke the corresponding extension dispatcher[4].

As mentioned earlier, the core server remembers if byte swapping is necessary for a specific client's packets. If byte swapping is necessary the core server will invoke the SwapDispatcher. Otherwise, the NoSwapDispatcher will be activated. In either case the dispatcher should refer to the *MinorOpcode* in the packet and dispatch the packet to the corresponding routine to perform the desired operation. Figure 5 shows an example of the extension dispatching scheme. In this case the extension is called *PEX*. From the MajorOpcode the core server determines that the received packet is for the PEX extension. Depending on the byte ordering of the client, *SProcPEXDispatcher()* (SwapDispatcher), or *ProcPEXDispatch()* (NoSwapDispatcher) will be invoked. This example shows an efficient way to implement the dispatcher by using the MinorOpcode as an offset into a procedure jump table to invoke the corresponding function. Normally, the swapping routine will simply swap the received packet and call the non-swapping routine using the byte swapped packet. In the given example, the swapping routine is *xPEXSProcCreateLookupTable()*. After swapping, it calls the non-swapping routine, *xPEXProcCreateLookupTable()*, to perform the desired operation.

Reply Scheme. As in the request case, the extension server depends on the core X server for maintaining the byte ordering consistency with the client. Figure 6 uses the PEX extension as an

[3] *ResourceType* is created by the core X *CreateNewResourceType()* routine, and *ResourceClass* is created by the core X *CreateNewResourceClass()* routine. Please refer to the server extension document [20] for the details on resource classes and types.

[4] This would be one of the two dispatchers registered with the core X server during the extension registration.

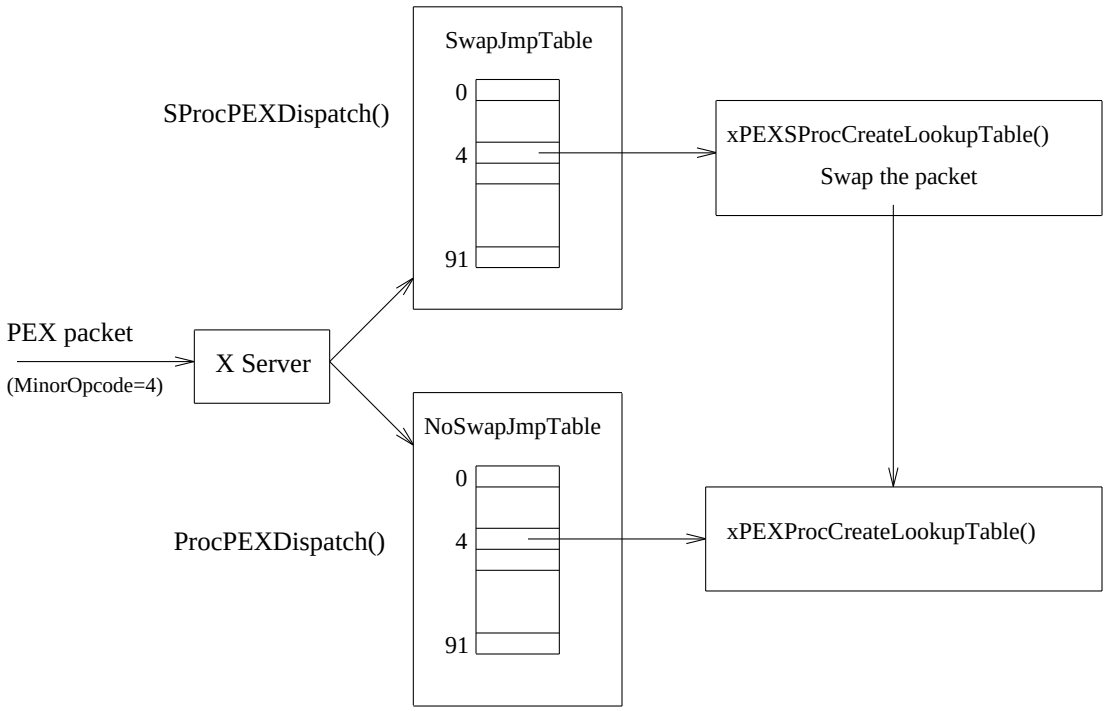

Figure 5: Extension Dispatching Scheme

example. The extension server passes the packet and a swapping procedure-pointer to the core server. The core server will decide if the extension server's swapping routine should be invoked.

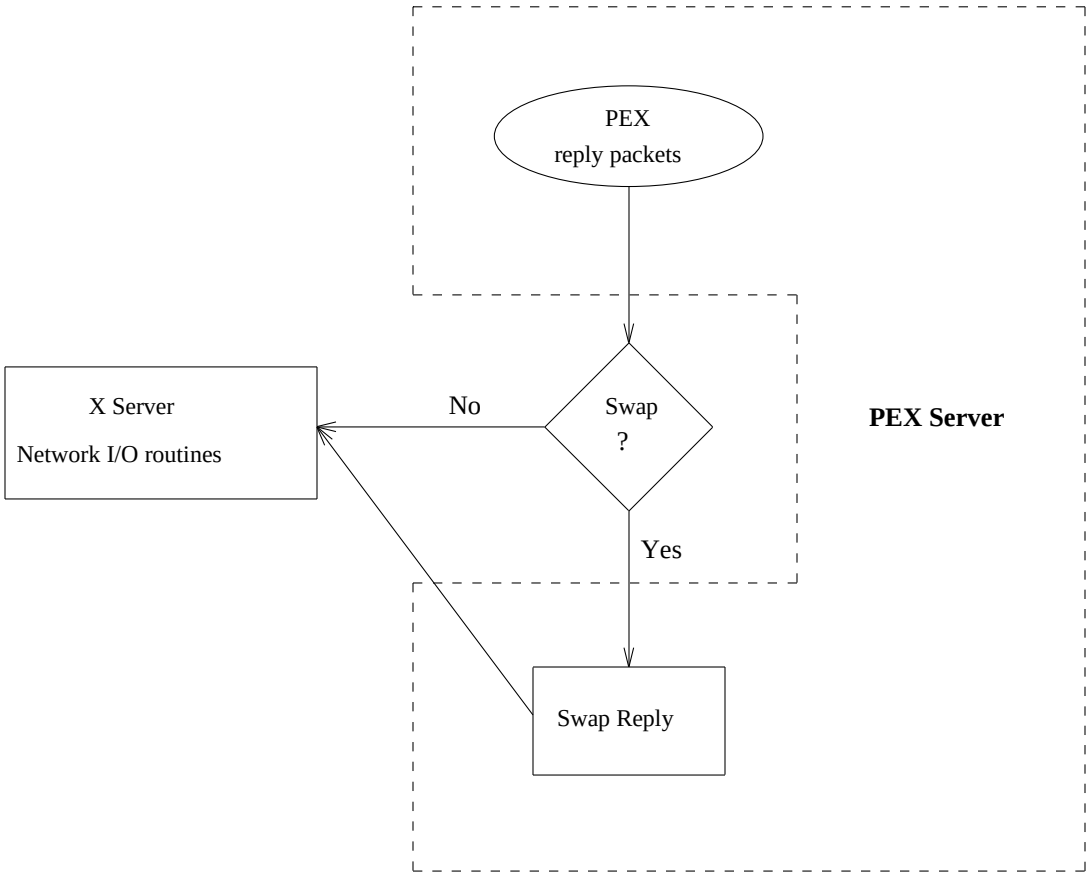

Figure 6: Server Reply Scheme

Event Sending. The extension handles events in exactly the same way as it handles replies. A different core X server routine is invoked to send event packets. The mechanism is the same as in figure 6.

3.5. Operational Model

The previous two sections discussed the responsibilities and operation of the major functional blocks in building an extension to X. This section provides a summary describing how these blocks cooperate, and the coordination between the client extension library and the extension server.

When the server begins execution, the extension server must register itself with the core server by

calling the AddExtensions() routine. The extension specific information (error codes, event codes, and major opcode) are assigned to the extension server during the registration. The client extension library must query the server to obtain the extension specific information before it can access any of the functionality provided by the extension server. The client side extension library is responsible for filling the request packet headers with the extension specific information to indicate that the packets are for the extension server. The core server examines the MajorOpcode of the received packets and forwards the packets to the extension server by invoking one of the registered dispatchers. The extension server sends errors and events to the client with valid codes in the assigned range.

Since the server is not supposed to enforce policies, the extension server should be viewed as an additional functional box. The client side extension library can be considered to be the interface to the additional server functionality. The client applications and libraries are responsible for coordinating all the functional blocks in the server in their performance of the desired actions.

3.6. Existing X Extensions

There are many extensions implementing different functionalities in the X environment. The application domain of these extensions varies widely from real time video display [23], to PostScript support [24], to 3D graphics [8]. The existence of these extensions attests to the effectiveness of the extension mechanism. It also shows that the basic X philosophy of providing a simple, useful and efficient environment for the development of special extensions, is a good design approach.

The major difficulty in developing an extension to X is the lack of complete documentation. Designers must gather information from many different sources (including the source code). Also, the existing documentation on extension server development is out of date.

CHAPTER 4.

PHIGS and PHIGS PLUS

The *Programmer's Hierarchical Interactive Graphics System* (PHIGS) has been adopted by the *American National Standard Institute* (ANSI) as a standard for 3D computer graphics [15]. It inherited some basic concepts from the *Graphical Kernel System* (GKS), the ANSI adopted 2D computer graphics standard [19]. The fundamental goal of PHIGS is to allow the development of well defined, portable, and device independent 3D graphics applications. Besides extending the graphical capability of GKS, PHIGS also provides a better modeling environment. This chapter describes the PHIGS system model, design philosophy, and the motivation for the proposed *PHIGS PLUS* standard [16]. After this discussion, it will be clear that some problems arise in integrating PHIGS+[1] with an existing 2D windowing environment.

4.1. PHIGS System Model

PHIGS can be separated into *Graphics* and *Modeling* sub-systems (see figure 6). The modeling component provides the facility for the applications to describe the objects to be displayed. The graphics sub-system of PHIGS is basically the same as that of GKS. It allows the application to display the objects from the modeling sub-system in a device independent manner. *Structure Traversal* is the process that links the two sub-systems together. The traversal process examines the modeling information and sends commands to the graphics sub-system to generate an image on the display.

4.2. The Modeling Sub-System

PHIGS maintains a *Centralized Structure Storage* (CSS) to separate the modeling from the basic graphics operations. An application typically edits the structures in the CSS to model objects and then displays these objects by *posting* the CSS structures. A CSS structure is composed of a sequence of elements organized into a list. Structure elements can be categorized into: output primitive, output primitive attribute, coordinate transformation, or execute structure.

> **Output primitives** are elements that, during traversal, cause the rendering of primitive geometric information onto the display device. Valid PHIGS output primitives are: Polyline, Polymarker, Text, Annotation Text, Fill Area, Fill Area Set and Cell Array.

[1] From here on, *PHIGS+* will be used to refer to the combination of PHIGS and PHIGS PLUS.

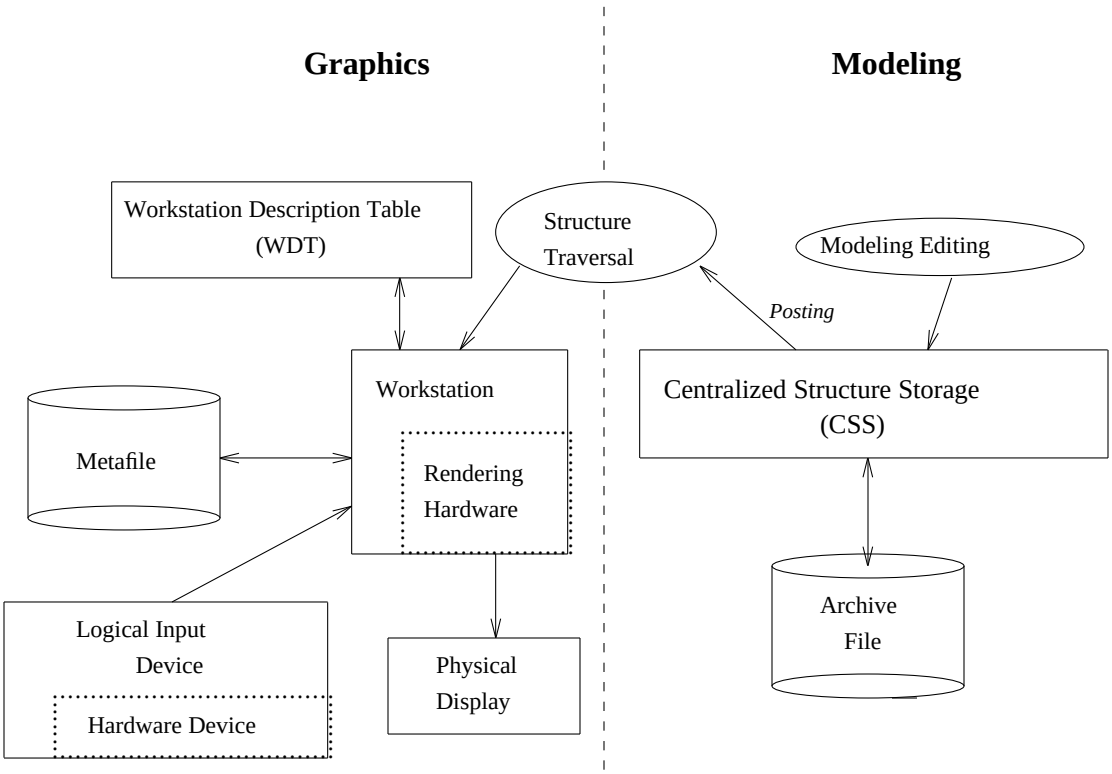

Figure 7: PHIGS System Model

Output primitive attributes are elements that change the appearance of the output primitives. For example, some of the valid attributes for Polyline are LineWidth, LineType, and LineColor.

Coordinate transformations perform two major tasks: they facilitate the definition of models, and they manipulate the orientation of the displayed objects. Some of these elements allow the definition of a device independent coordinate system. With this ability, the application can define 3D objects more conveniently. Coordinate transformation elements also allow the application to display the defined objects from different viewpoints.

Execute structure element causes the traversal of another structure in the CSS. This is analogous to a subroutine call in a high-level programming language. After the target structure is traversed (the subroutine execution is finished), the rest of the elements in the current structure will be traversed (control is returned to the caller). Execute structure element is very useful when it is desired to display the same object multiple times. For example, an automobile wheel can be defined into a CSS *Wheel Structure*. The CSS structure of a car can then execute the *Wheel Structure* four times with different transformations to display the four wheels. Since

cycles in the execute structure chain cannot exist, the structure hierarchy thus formed is a directed acyclic graph (DAG).

The PHIGS application defines objects by editing the list of structure elements in a CSS structure. Valid editing operations are *Insert*, *Delete*, and *Replace*. PHIGS defines the *archive file* for long term modeling storage. After defining the objects in the CSS, an application can store the objects to the external file system via the archive file. When desired, the application can retrieve the archived objects from the external file system into the CSS.

The CSS structures are not displayed on the workstations until they are *posted*. Only posted CSS structures can be visible. Posting a structure causes the structure to be *traversed* and thus the image of the structure is generated on the workstation. Structure traversal is the process that goes through the list of structure elements in the CSS structure. Each element is examined and a request is sent either to the *Traversal State List* or to the graphics sub-system. Attributes modifying elements are interpreted by setting the appropriate values in the Traversal State List. Structure traversal sends *Output Primitive* elements to the graphics sub-system to generate output. The graphics sub-system refers to the Traversal State List for attributes that control the appearance of the primitives. For example, if a CSS structure contains the following elements:

SetLineColor(Red)
SetLineWidth(3)
SetLineType(Solid)
OutputPolyline(Line_A)

the application could post this structure to invoke the traversal process. The structure traversal would process the first three elements by sending requests to the Traversal State List to set the current line color attribute to red, the current line width to three-pixels wide, and the current line type to solid. The OutputPolyline element would then be sent to the graphics sub-system, and Line_A would be displayed as a three pixel wide, red, solid line.

4.3. The Graphics Sub-System

The graphics sub-system is based on the concept of a graphical workstation. It encapsulates and provides an abstract interface to the input and output capabilities of the underlying hardware. The *Workstation Description Table* (WDT) describes the capabilities of the physical device. Workstations are categorized as *INPUT*, *OUTPUT*, or *OUTIN* to capture different device capabilities. All workstations in the OUTPUT and OUTIN categories have the capability of processing output elements (sent from structure traversal) to render pictures on the physical display device[2]. PHIGS defines bundle tables to support *workstation dependent attribute* information[3]. The *Aspect Source Flag* (ASF) is the mechanism that arbitrates the attribute control between the workstation and the modeling sub-system. For example, an application can set the Line_ASF to allow the displaying workstation to control the line attributes. In this case, instead of getting line attribute information

[2] The *Display device* can be any output device. For example, raster color display, plotter or printer are typical output devices.

[3] Since *workstation dependent attribute* information cannot be changed during structure traversal, it is also referred to as *static modeling information*.

from the Traversal State List, the information will be obtained from the workstation LineBundleTable. The workstation appearance is tightly controlled by the application. PHIGS provides a set of elaborate appearance control options to allow the application to maintain the workstation display consistency[4]. For example, when a posted structure is deleted from a workstation, the application has the option to regenerate the entire display screen, simulate the effect, or to not perform any action.

4.4. PHIGS PLUS

PHIGS is a 3D extension to GKS because it defines the GKS primitives (2D) in a 3D environment. There are no 3D primitives (surfaces) or primitives that facilitate 3D modeling (Triangle Strips, Quadrilateral Meshes). *PHIGS Plus Lumiere und Surfaces* (PHIGS PLUS) [16] is a proposed ANSI standard to extend the PHIGS definition to better support 3D graphics applications. The important thing to notice is that the PHIGS PLUS definition does not change the semantics of PHIGS; it merely introduces new structure elements and expands the static modeling information of the workstation.

PHIGS PLUS establishes a mechanism to model surfaces and curves. Some compound output primitives like *Triangle Strips* and *Quadrilateral Meshes* are defined to facilitate 3D modeling. The reflection attributes of these primitives under different light sources are also defined. The defined reflection attributes are: ambient, diffuse, specular, and transmission. PHIGS PLUS defines three different shading methods: Phong, Gouraud and Dot Product [25]. The workstation dependent attributes are expended to support a light source table, a depth cue table, and a primitive edge bundle table. PHIGS PLUS defines four different light source types: ambient, positional, directional, and spot light.

of15 6(ev)X 24546(is)X 397(cos60(d8)X 48 352962 X)X 367llow)abspsende341(c684 47((G8 29)X 3545(7(G(thi325 (server)1(re

4.5. PHIGS+ Design Philosophy

PHIGS+ makes the assumption that its application is the only process performing output on the underlying physical device, and thus has total control of the environment. All system resources are considered to be exclusively owned by the PHIGS+ application and can be altered at will. *Internal* control [22] is the user interface model adopted by PHIGS so that the application is always in charge of the control flow. When desired, the application can initialize an input device and wait for the user's response: again, control always remains with the application. In the PHIGS+ definition there is no notion of synchronism or cooperative processes (as in a distributed environment). As explained in Chapter 6(on)X 1637(X156endsThaN 1016(bystem)X 2haviexch def 12(Ffle40871825_A)X40(f160(864 .)X 64erver)X