1

Introduction

This document describes the architecture of the PEX Sample Implementation (PEX-SI). PEX-SI has been developed by a team of software engineers at Sun Microsystems in Mountain View, California. This document describes the components of the sample implementation's design, its architecture and features, and some functional details of the inner workings of its modules. Finally, it gives pointers to other related references.

This document is not intended to serve as an aid to understanding PEX protocol and encoding, the PHIGS and PHIGS PLUS standards, or the X Window System. It is assumed that the reader has a fair knowledge of the terms and the underlying technology of all three.

1.1. Audience

The intended audience for this document consists of PEX application developers and X Window System programmers interested in achieving 3D graphics within an X environment. Programmers who might be porting PEX-SI to a specific environment should also refer to the *PEX-SI Porting Guide*.

1.2. Scope

This document describes the following aspects of PEX-SI:

- Client Side Overview and Architecture
  - Application Programmer's Interface (API)
  - PHIGS Monitor (PM)
  - PHIGS Central Structure Store (CSS)
  - Workstations and Workstation Types
  - Window Event Processing
  - PHIGS Input Queue Handling
  - Archival Support
  - Double Buffering
  - Error Reporting
- Server Side Model Extension (ME) Overview and Architecture
  - diPEX Implementation
  - ddPEX Interface and Implementation

- Resource Management and Dependencies
- PHIGS Workstation Module
- Shared Resources Module
- Rendering Control Module
- Transformation Module
- 3D Rendering Module

1.3. Overview of PEX-SI

PEX-SI is a sample implementation of support for the complete PEX protocol as defined in the *PEX Protocol Specification ver. 5.0P* and the *PEX Protocol Encoding Document ver. 5.0P*. The primary goal of PEX is to provide 3D graphics capabilities based on PHIGS and PHIGS PLUS policies, within the policy-free X Window System model. To achieve these goals, PEX-SI provides the following:

- A PHIGS/PHIGS PLUS Application Programmer's Interface (API) for clients that support PHIGS/PHIGS PLUS policies through the mechanisms defined by the PEX protocol
- A Model Extension (PEX-ME) to the X sample server that processes PEX protocol requests

PEX-SI has three major goals:

- Quick and easy handling of PEX requests
- Quick and easy integration into existing X servers
- Easy adaptation to a wide range of client and server platforms

Client-side support in PEX-SI consists of the API and an optional PHIGS Monitor (PM). The API supports a PHIGS/PHIGS PLUS C language binding. The PM is a separate process that is started by the API. The PM manages PHIGS input, client-side structure storage, and X window events. The API and the PM generate the X and PEX requests required to support PHIGS and PHIGS PLUS within the X environment.

Server-side support in PEX-SI consists of a Model Extension (PEX-ME) to the sample X11R4 server. It follows the guidelines for implementing an extension to X described in the *Definition of the Porting Layer for the X v11 Sample Server*. The sample X server architecture has three functional areas: the operating system-dependent code (os), the device-independent code (dix), and the device-dependent code (ddx). Similarly, PEX-ME consists of operating system-dependent code (osPEX), device-independent code (diPEX), and device-dependent code (ddPEX). To ease porting, the ddPEX code is further modularized, as described later in this document. The complete PEX-ME generates output to simple frame buffer hardware via the capabilities of X11R4's ddx.

Each of the main sections of the PEX-SI code (API, PM, diPEX, ddPEX, osPEX) is designed to be as independent as possible, with a clear, understandable, and well-defined exchange of data between the sections. This facilitates porting and tailoring of the code for a wide variety of PEX implementations and graphics devices.

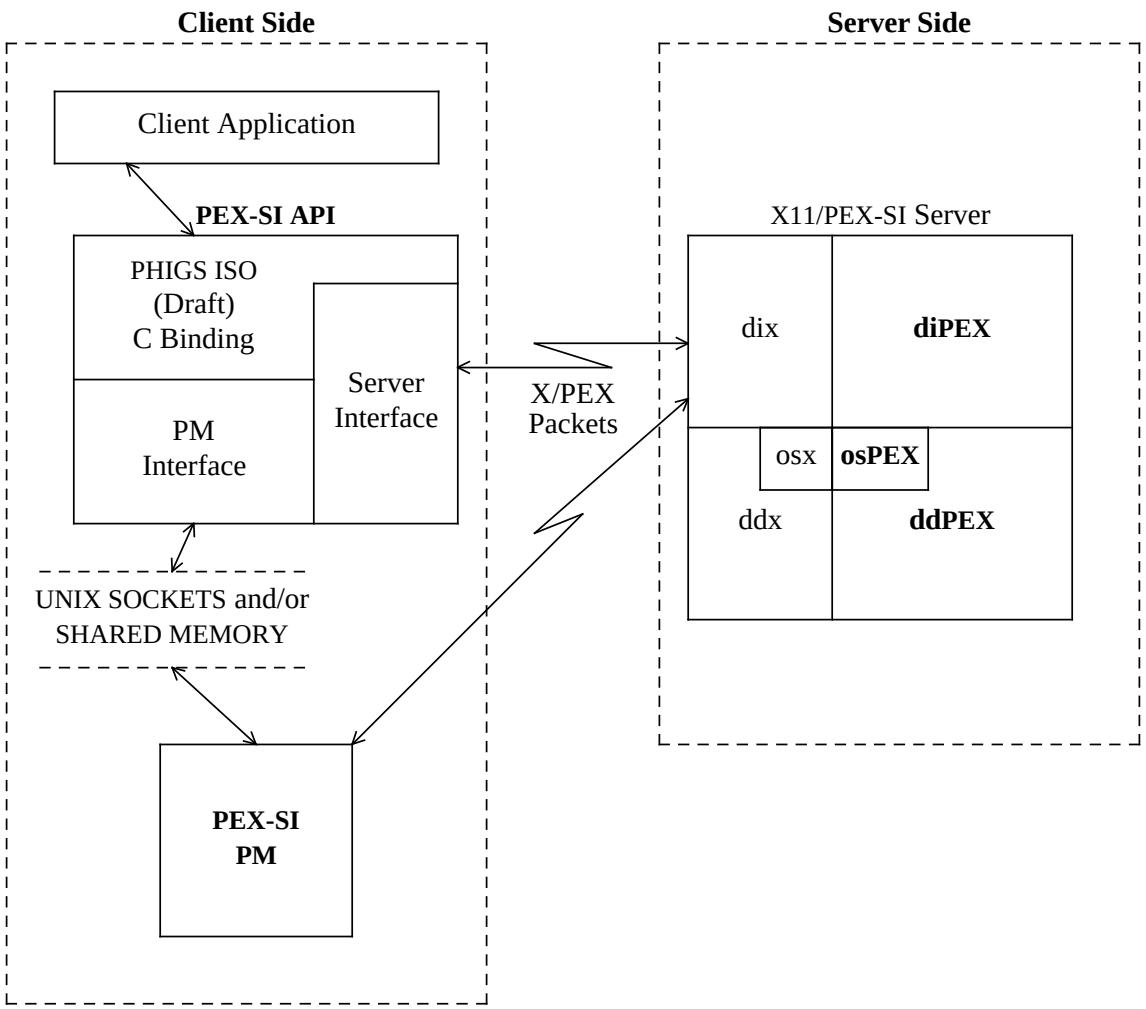
Because PEX is a two-process model, and because PHIGS has the concept of a Central Structure Store (CSS), PEX-SI has to deal with the issue of ''where'' to implement the CSS.[1] The PEX protocol allows the CSS to be implemented in either the client side (client-side CSS), or the server side (server-side CSS). The PEX protocol also defines client-side traversal, server-side traversal, and mixed traversal. Client-side traversal occurs on the client side and sends output commands to the server. Server-side traversal occurs in the server. Mixed traversal allows one client to use both client-side and server-side traversal methods.

[1] See the *PEX Introduction and Overview* for a complete description of structure storage models.

PEX-SI implements both CSS models, and both client and server-side traversal. PEX-SI does not support mixed traversal.

Figure 1-1 presents a high level view of the processes comprising a simple PEX-SI user environment. The data exchanged between the client and server is transmitted and received via X and PEX protocol packets. This diagram shows the general data flow between the client and server. Communication between the client and server involves much more than generating the simplified data shown in this figure. The rest of this document and the *PEX-SI Porting Guide* describe the client and server code design and structure in more detail.

Figure 1-1 *High-Level Diagram*



The client application communicates with the PEX-SI library through the PHIGS/PHIGS PLUS C binding in the API. The API maps PHIGS library function calls to the appropriate X, PEX, or PM requests, and transmits the requests to the PM and the server. The API and the PM use Xlib to generate X requests when necessary. The PM uses Athena widgets to realize some input devices. API calls to the PM are similarly mapped to X or PEX requests by the PM. The PM also intercepts X events; it generates X and PEX requests to respond to the events and relays information back to the API when and as needed.

All requests coming to the server are received by dix, which determines if the request is an X or PEX request. X requests are handled by dix and ddx in the usual manner. PEX requests are passed to diPEX. diPEX converts the

requests to the correct (server native) byte-ordered form and floating point format, and calls into ddPEX, which then processes the request. For requests that cause graphics to be rendered, ddPEX eventually passes the primitives (in device coordinates) to ddx to be displayed.

The data exchange between diPEX and ddPEX is described in greater detail in Chapter 3, The PEX-SI Server Side.

1.4. Porting PEX-SI

PEX-SI is designed to be portable to a wide variety of software and hardware environments. PEX-SI defines portability as a balance of flexibility and adaptability. *Flexibility* is achieved when the code at every level is either documented well enough or is obvious enough in its execution that it can be worked with easily. *Adaptability* is achieved when the code can also be optimized to perform well in a wide variety of environments.

Although it uses the fundamentals of the X11 architecture as a model, PEX-SI expands upon that model in its approach to portability. The X11 architecture addresses an audience of users interested in creating original applications for the X protocol, and hence it concentrates on making the client interface as portable as possible. In addition, PEX-SI addresses an audience that is interested in adapting existing PHIGS applications to work as clients via the PEX protocol over a network. Where X11 only provides for server porting at either the highest (ddx) or lowest (pixel read/write) levels, the PEX-ME has been designed to allow ease of porting to a greater range of devices with a diverse range of capabilities. Thus, while X11R4 achieved its greatest level of portability in the client-side code, PEX-SI has also achieved a high level of portability in its server implementation.

To accomplish this level of portability PEX-SI provides code that both runs efficiently in a specific implementation in a simple frame buffer environment, and lends itself well to performance improvements achieved by modifying or replacing the internal layers, or portions thereof, of the PEX-SI code. The PEX-SI client-side and server-side code is divided into distinct layers explicitly designed to ease porting.

The client side code is organized in such a way that the toolkit, operating system, and PEX protocol-dependent portions are separated and isolated into distinct modules and directories. PEX-SI presents the user with two workstation models that allow applications to be independent of the toolkit used, or, alternatively, to be closely tied to a particular toolkit thereby enhancing performance. The creation of the PHIGS Monitor (PM) provides applications the ability to fully handle asynchronous communication with the server. The PM is optional, however, suiting the requirements of synchronous operating systems. In addition, the API is designed to utilize shared memory and/or UNIX sockets to accomodate both System V and BSD UNIX, and non-UNIX systems. Finally, the API code that uses the PEX protocol is isolated in a single directory. The majority of the API code is PEX independent.

PEX-SI also provides several distinct porting layers in the server, allowing for a wide range of possible performance improving ‘‘short cuts’’. The portions of code most likely to be changed to improve PEX-SI performance in a particular environment (i.e., operating system or device-dependent structures) have been isolated into separate directories and files. In addition, the device-dependent code has been further layered so that individual pieces can be independently modified or replaced with implementation-specific software or hardware interfacing components.

PEX-SI is designed to run on a platform judged to be the least common denominator of all PEX Consortium sponsors: the simple frame buffer. Thus, PEX-SI can be used as soon as it is installed; there is no requirement for any initial porting, however, most users will want to customize and optimize the existing code. In some cases this will mean adjusting compilation variables; in other cases, it will mean complete replacement of certain modules of code. To this end, the *PEX-SI Porting Guide* provides the specific information required to customize and optimize PEX-SI for a specific environment. Refer to that document for specific implementation details. The remainder of this document discusses the architecture of PEX-SI at a higher level.

1.5. Related References

PEX Introduction and Overview (v 3.2; 10 October 1988): an introduction to the goals and structure of the PEX project, with a glossary of PEX terms. *It is widely recognized that this document is out of date.*

PEX Protocol Specification (v 5.0P; 14 September 1990): a description of the protocol extensions to the X Window System that comprise PEX.

PEX Protocol Encoding Document (v 5.0P; 14 September 1990): the bit bindings of the PEX protocol.

X11 Server Extensions (28 August 1987): a description of the interface between the MIT X11 Sample Server and portable extensions in general.

Definition of the Porting Layer for the X v11 Sample Server (1 March 1988)

Xlib — C Language X Interface (X v11 R4)

X Toolkit Intrinsics — C Language Interface (X v11 R4, March 1988)

ISO/IEC 9592-1:1989(E), Information Processing Systems — Computer Graphics — Programmer's Hierarchical Interactive Graphics System (PHIGS), Part 1: Functional Description

Draft proposed PHIGS PLUS, ISO/IEC JTC1 SC24-N454 (20 March 1990)

PEX-SI User Guide (29 February 1991): introductory information about the PEX-SI clients, the API, and PEX-ME; instructions about installation and using the clients.

PEX-SI Graphics Library Manual Pages (29 February 1991): reference manual for the API.

PEX-SI Porting Guide (29 February 1991): detailed description of PEX-SI with an emphasis on porting portions of PEX-SI to various environments.

2

The PEX-SI Client Side

2.1. Client Side Overview

The PEX-SI client side is made up primarily[2] of the Application Program Interface (API) and the PHIGS Monitor (PM). The goals and constraints that led to this design, and the resultant architecture, are described in this chapter.

The Application Program Interface to PEX is provided by the PHIGS and PHIGS PLUS C bindings. The API is based on the IS version of the binding. The PHIGS PLUS C binding effort has just begun in ISO, so there is no working draft. The API has developed an interim version to use until a working draft (or better) is available. This version follows the same conventions used by the DIS PHIGS binding.

All instances of ''PHIGS'' in this document refer to both PHIGS and PHIGS PLUS unless otherwise specified.

2.1.1. Goals

The primary goal of the API is to provide a programmatic interface to PEX. A number of other goals drive the design, however:

- The implementation of a conforming and efficient PHIGS using X and PEX.
- To the extent possible, the smooth running of the PHIGS application in the window system, with no restriction on the window system operations the operator can perform on the PHIGS workstation windows.
- The support of multiple simultaneous workstations. These workstations can exist on any combination of servers with which the client can connect.
- The targeting of two broad types of PHIGS applications: those that do not have or want any window system knowledge and those that do. The former is a *vanilla* PHIGS application that does not know it is running in a window system. The latter is a window-system-literate (or toolkit-literate) application that wants to use PHIGS in conjunction with its own use of the window system.
- The provision of PHIGS output and input to both application types. The incorporation of optimizations for those applications that only require output.
- The shielding of the vanilla PHIGS application from the window system, but still using the window system to its advantage.
- Allowing window-system-literate applications to use the window system in conjunction with PHIGS, without losing any PHIGS functionality or conformance.

---

[2] PEX-SI also provides a number of clients that use the PEX-SI API to demonstrate its effectiveness in the sample implementation. The PEX-SI clients are not described in this document. Descriptions of the PEX-SI clients can be found in the *PEX-SI User Guide*.

- Providing default redraw and resize event responses that are transparent to the application, but allowing the disabling of this automatic response if the PHIGS client so chooses.
- Implementing PHIGS policies within the ‘‘policy-free’’ environment of X.
- Providing some limited support for synchronizing PHIGS API output with X events delivered by the X server to the application.

2.1.2. Constraints

The PEX-SI API was designed under the following constraints:

- To enhance portability, features not widely available should not be used.
- The API should not compete with the application for resources, services, or process control. The API assumes that a window-system-literate PHIGS application will be monitoring X events, either directly or through a tool kit. In order to avoid competition for events, the API cannot monitor events on the same stream (server connection) as the application. The API should not preempt use of any operating system or toolkit functionality.
- The API only has the flow of control infrequently (when the application calls a PHIGS function). The API cannot count on regaining the flow of control from the application at any particular point in time.
- Both the API and the PHIGS input devices need access to the PHIGS event queue. The API needs this access to inquire about the queue’s state and to retrieve events. The input devices need access to place events in the queue and to determine if the queue has overflowed. (The input devices must indicate to the operator whether or not events from the devices are successfully placed in the queue.)
- PHIGS locator, stroke, and pick input device events need to be resolved immediately so that the PHIGS event corresponds to the state of the workstation and Central Structure Store when the event occurred. These events cannot be resolved when they are removed from the queue by the API or the application — by then it will be too late to resolve them correctly.

2.2. General Architecture

To meet many of the above goals and constraints, it is necessary to have two separate threads of control: one for the application and one for the API. The API thread of control allows the API to manage input device events, the PHIGS input queue, and window events independent of the application. To achieve separate threads of control, the PEX-SI API is implemented in two logical parts — one residing in the client’s process and one residing in a separate process. This separate process contains the PHIGS Monitor (PM).[3]

The Client Side API code consists of the binding and an interface layer to the PM and the X/PEX server (PEX-ME). The interface layer coordinates the various requests and replies to the PM and to PEX-ME. It issues the PEX protocol for most API functions and keeps the PM informed of appropriate state changes.

---

[3] Another means of realizing two separate threads of control is through the UNIX signal handling mechanism. The second of the two threads is invoked upon the receipt of a signal (usually SIGIO) generated by some input device or pending input on the X event stream. While this method may work, it has a number of problems associated with it that can be avoided with the two-process scheme. Among these problems are the non-portability of performing memory allocation during signal handling, the additional logic required to detect when interrupts have occurred in normal processing and respond accordingly, the preemption of a portion of the signal handling mechanism (the API takes away the ability of the application to trap certain signals, most likely SIGIO), and the potential conflict of both the application and the API reading X events from the same event stream.

PEX-SI has weighed the advantages and disadvantages of both methods and has chosen the two-process method. Its advantages are that it avoids the above problems, does not preempt any use of operating system, window system, or toolkit functionality, can take advantage of parallelism on multi-processor machines, allows input and window events to be monitored and managed through conventional means, and allows the most flexibility with regard to changing input device realizations and window event response policy.

The PM performs the following:

- Processes window system and input device events associated with a client's PHIGS workstations.
- Implements the window system event response policy.
- Issues the required PEX and X calls to enforce that policy.
- Maintains the PHIGS input event queue, and services event requests and inquiries from the client-side API code.
- Manages the Central Structure Store for PHIGS workstations that exist on servers not supporting structure storage (see Section 2.3 for more information).

There is one PHIGS Monitor for each PHIGS client. The PM monitors all PHIGS workstations opened by a particular client regardless of what server each workstation is on (one PHIGS client may have workstations open on more than one server). The PM is started (via UNIX *exec(2)*) when OPEN PHIGS is called and is terminated when either CLOSE PHIGS is called or the application terminates. The PM opens server connections as workstations are opened, using a single connection to servers containing multiple PHIGS workstations. The PM process always runs on the same machine as the client runs on.

2.2.1. Applications with No PHIGS Monitor

If it is known by the application that no PM functionality is required (no PHIGS input, no client-side CSS, and no API handling of window events), the execution of the PM can be prevented by utilizing a flag specified in *popen_pex()*. For more information, refer to the *PEX-SI Graphics Library Manual Pages*.

2.2.2. Communication Between API and PM

All communication between the client-side API and the PM is initiated by the client code; the PM does not issue asynchronous communication to the client-side API. Any communication from the PM is read by the client-side API before control is returned to the application.

The PM maintains only the state it needs to perform its duties. The PM is not informed of all changes in PHIGS state, nor is it necessary for the API to communicate with the PM during every PHIGS function call. Except when the PM is maintaining the client-side structure store, the API communicates only changes in input device state, some changes in workstation state, and input-related requests. However, when the PM is maintaining the client-side structure store, the API communicates with the PM during the majority of PHIGS function calls. In this case, the API to PM protocol is dense; usually one PHIGS function call causes one communication packet to be sent to the PM.

Communication between the PM and the API is through shared memory and a bidirectional socket. If shared memory is not available, then all communication is through the socket. When shared memory is available, the socket is used only for synchronization between the client and the PM and to communicate blocks of data too large to fit in a fixed-size shared-memory buffer.

2.2.3. API/PM Communication Protocol

The protocol used between the client-side API and the PM is described by the structure definitions in the files *phgargs.h* and *phgretdata.h* in the API's private include directory. These *phgargs.h* structures consist of an opcode, a fixed-size portion, and a variable-size portion. The ''args'' structures are written to the PM by the client-side API code. If shared memory is available, the structures are written directly to a queue in shared memory. If shared memory is *not* available, they are written to the socket. The PM reads the structure contents from either location and performs the indicated operation. If a reply is required, the PM writes the reply to either a location in shared memory or to the socket. The client-side API reads the reply data from either location. Synchronization of access to the shared memory buffer is via a lock field in the buffer. Because the client-side API and the PM run on the same machine, there

are no special data format considerations. The protocol is not intended to be used across a network, between dissimilar machines, or between dissimilar versions of the client-side API and PM.

2.2.4. Error Reporting

The client-side API and the PM use an additional socket for error reporting so that all errors can be reported by the client-side code via the PHIGS error-handling mechanism. All errors detected by the PM are sent to the client-side API through this error socket. The PM does not report any errors directly. (See Section 2.7.2 for a discussion of the synchronization issues associated with this.)

2.2.5. Control

The client-side API does not rely on regaining control of the application, nor does it rely on the application issuing an X request that will cause an X connection's buffer to be flushed. The client-side API ensures that all X and PEX requests necessary to service a PHIGS function call are sent to the server prior to returning control to the application.

2.2.6. Window Events

The PM ensures that no PHIGS state that is visible to the application is changed when it initiates a redraw. The exception is that changes to the Central Structure Store made subsequent to the last update of the display surface may be reflected in the redraw.

Because the API receives responses to window system events from the PM, the application cannot replace the window system event response functions. The application can only prevent their execution, in which case the application assumes responsibility for responding to the events. Application developers may, however, implement and supply their own specific PM to replace the event response functions.

The application uses the workstation description table modification functions (*phg_ws_type_set()*, etc.) and PHIGS escapes to control some of the interaction with the window system. There is an escape to disable and enable the PM's handling of window events. If the application does not want the PM to respond, then it is presumably monitoring the window events itself and responding as it sees fit.

The following describes PM responses to some window system events:

Window Move No response.

Window Resize PHIGS retraverses the posted structures, clipping to the new window size.

Window Close No response.

Window Exposed Redraw exposed portions of window by retraversing, but clip to only the exposed regions. The exposed portions may be more up-to-date than the unexposed portions if the workstation has not been updated because changes were made to the CSS or workstation state list.

Window Occluded No response.

Window Destroy The PM exits by calling exit(3) with a value of 3. This also causes the client process to exit with a value of 3.

2.2.7. PHIGS Input Queue

The PHIGS input queue is maintained in the PM. Applications retrieve PHIGS events from the queue by calling *pawit_event()*. This function retrieves the event at the head of the queue, if any, and puts it in the current event report in the client. The event is read from the current event report with the appropriate GET <class> function for the event type. If the PM is not used, *pawait_event()* returns immediately without waiting for the timeout period specified in the

function call.

The PHIGS event queue in the PM holds 500 events. If an attempt is made by the operator to place a PHIGS event in the queue when the event queue is full, the PM calls the Xlib function *XBell()*.

2.2.8. Archival Support

The archive file format used by PEX-SI is used for the storage and subsequent retrieval of PHIGS and PHIGS PLUS structure elements. The file format conforms to the required archive file specification in PHIGS as to form and functionality of a generic set of archive element types. However, since the PEX archive file format is not a registered PHIGS format, it is classified as a private encoding. This encoding may be submitted for ANSI registration at some time in the future.

The following guidelines were used in the design of the archive storage format:

- Define structure elements using the PEX protocol encoding.
- Conform to the PHIGS archive file specification.
- Allow all PEX binary representations to be supported (written and read).
- Allow efficient support of devices that provide random access.

The PEX protocol already encodes PHIGS and PHIGS PLUS structure elements for use in a byte stream representation. The PEX archive format uses this same encoding format for all structure elements.

The PHIGS archive interface supports any of the binary representations that the PEX protocol can support. By reformatting the binary values, implementations can read and write files that are not written in the host computer's native format. However, it is not intended that archives of different binary representations be allowed within one archive file.

The archive file format supports random access devices completely. It also allows sequential access, if necessary.

For more information on PEX-SI archival support, refer to Appendix C of the *PEX-SI User Guide*.

2.2.9. No Metafile Support

PEX-SI does not support metafiles.

2.3. Central Structure Store

There are three structure storage situations supported by the API:[4]

1. All open workstations are associated with servers supporting the ''PHIGS workstation subset.''
2. One or more workstations are associated with a server that supports only the PEX ''immediate mode rendering subset.''
3. There are no open workstations.

In the first case, the API broadcasts most PEX requests to all the involved servers; for example, structure editing operations are sent to all the servers. In this case, the CSS is duplicated on all involved servers; there is no attempt to store only those structures posted to workstations associated with a given server.

[4] Actually there are four — the fourth being the ''Mixed Traversal'' case. This case is not directly supported by PEX-SI.

In the second case, the PM contains a CSS for all the servers not supporting workstation resources and structure storage. This copy of the CSS is treated similarly to those of servers supporting structure storage in that it contains a complete copy of the CSS and is updated with each relevant PHIGS function call. The PM contains the PHIGS workstation logic and provides the application with the PHIGS workstation functionality.

The third case is the situation immediately after PHIGS is opened, but no workstations have yet been opened. Because PHIGS CSS operations can be performed regardless of whether any workstations are open, there needs to be a CSS independent of any open workstation. This CSS exists in the *default server*. The default server is specified prior to or during the call to OPEN PHIGS. If the default server does not support structure storage, or there is no default server, a CSS will be created in the PM.

The default server is generally expected to be the local server. In many cases the local server will support structure storage and workstation resources, and all workstations will be opened on that server. In such a case only one CSS will be built and maintained. If the local server does not support structure storage or PHIGS workstations, there will still be only one CSS, but it will reside in the PM.

The API will detect which of the above cases exists at any time and react accordingly. For instance, when a workstation is opened on a new server that supports structure storage, the API will copy the existing CSS to that server. When a workstation is opened on a server that does not support structure storage, the API will create a copy of the CSS in the PM. Whenever all workstations are closed on a server other than the default, the CSS copy on that server, if any, will be destroyed. If there is a default server, it will retain a copy of the CSS for the duration of the PHIGS session — in other words, until CLOSE PHIGS is called.

2.4. Workstations and Workstation Types

The API supports two general types of workstations: a *tool* workstation and a *drawable* workstation. The major difference is that a tool workstation uses an API-created window and a drawable workstation uses an application-created window.

A workstation type in PHIGS is some unique identifier. In the API, the identifier is a pointer to a data structure. This data structure, which is not visible to the application, contains the Workstation Description Table (WDT) and other data associated with the workstation type. (Additional workstation information is also kept in a corresponding data structure in the PM.) The API augments the WDT to contain window-system-specific information in addition to the WDT information defined by PHIGS. The API-defined information describes additional characteristics of the workstation, such as window size and placement, and the environment in which it is operating. Most of the API-defined data in the description table can be changed by the application.

PHIGS defines two classes of workstation types: *generic* and *specific*. Generic workstation types are those that are not associated with an open workstation. The information contained in the WDT is a description of the desired characteristics and capabilities of a workstation of that type. In most cases, the WDT reflects the actual capabilities a workstation of that type would have if it were opened.

Specific workstation types are those associated with an open workstation. When a workstation is opened, the API makes a copy of the description table associated with the specified workstation type. This copy is then modified to reflect the actual capabilities of the display device and the characteristics of the environment. A new, specific workstation type is created, and the modified description table is associated with it.

2.5. Workstation Description Tables

As described previously, an augmented workstation description table (WDT) is associated with all workstation types. The contents of the WDT are mostly dependent on the display device on which a workstation of that type will appear. To emphasize this, the API associates generic workstation types with a particular display. The display name is stored in the augmented WDT. When a tool workstation is opened, it is opened on the display specified in the augmented

WDT.

When the API is initialized (i.e., when PHIGS is opened), the predefined workstation types are built using the capabilities of the default display.

2.6. Double Buffering

Double buffering in the PEX-SI API is controlled with the workstation type attribute *PHIGS_DBL_BUF_MODE*. This attribute sets the double buffering mode of the specified workstation type to the given value. A value of TRUE requests double buffering. A value of FALSE requests that double buffering not be performed. The API does not directly realize double buffering. If a workstation resource is used to implement the PHIGS workstation, the API just sets the double buffering mode of the workstation resource to the appropriate value. If a workstation resource is not used — for example, when client-side structure storage is being used — the API does nothing to realize double buffering. The onus is on the client application to control buffering via some exposed, compatible mechanism such as the proposed X Consortium multi-buffering extension.

2.7. Error Reporting

There are two classes of errors generated by the API: PHIGS errors and X/PEX errors. All PHIGS errors are reported using the standard PHIGS error reporting mechanism — a call to the function *perrorhand()*. The API attempts to trap some X and PEX errors generated by its use of X and PEX and report them via the PHIGS error-handling mechanism. Implementation-dependent PHIGS errors are defined for the X and PEX errors that the API could generate. When it is not possible for the API to trap X and PEX errors, they are reported via the X error-reporting mechanism — a call to the X error handler. The API does not register its own X error handler via *XSetErrorHandler()*.

2.7.1. Asynchronous Notification

It is not practical to keep all PHIGS state information in the portion of the API that lies in the application's process; therefore, some errors are detected in the remote portion of the implementation. Because performance is enhanced by buffering the processing of commands, and some errors are not detected until a command is processed, better performance can be achieved if notification of errors occurs asynchronously, such as during a function call made subsequent to the function that detected the error.

This asynchronous notification is the default behavior. Errors may not be reported before the detecting function returns. They will be reported during a subsequent function call, and the report will contain the correct function identifier — the identifier of the function detecting the error, not the one during which the notification occurs.

2.7.2. Synchronous Notification

A dynamically settable escape function is provided for the application to indicate that it requires synchronous error notification, so that notification of errors is performed before the detecting function returns. Because synchronous notification reduces API performance, it should be used only when the application requires it.

Any pending errors are delivered when the escape is set.

3

The PEX-SI Server Side

3.1. Server Side Overview

The server-side support in PEX-SI is a Model Extension (PEX-ME) to the X sample server. PEX-ME provides additional code that is linked with the X11 sample server and processes PEX protocol requests. PEX-ME follows the guidelines for implementing an extension to the X sample server as described in the *Definition of the Porting Layer for the X v11 Sample Server* (1 March 1988).

The PEX protocol defines two subsets of PEX that provide different mechanisms for generating 3D graphics. These subsets are defined in Appendix A of the *PEX Protocol Specification*. The ''PHIGS workstation subset'' of PEX provides the resources needed to support a PHIGS workstation, server-side structures, picking, and incremental spatial searching. This subset also includes a mechanism for traversing and rendering structure elements within the server. The ''immediate mode rendering subset'' provides the resources and rendering capabilities needed to support the immediate display of output commands.

The PEX protocol also specifies that the structure elements defined for the PHIGS workstation subset are identical to the output commands defined for the immediate mode rendering subset. In fact, structure elements can be thought of as stored output commands. A consequence of this specification is that structure elements can be rendered in the same way as output commands.

PEX-ME is designed to take advantage of the PEX subset definitions; it provides a mechanism for implementing both of the PEX subsets in a server, and at the same time maximizes the amount of code that can be shared between the subsets. Following the architecture of the X sample server model, PEX-ME is divided into three functional areas: device-independent PEX (diPEX), device-dependent PEX (ddPEX), and operating system-dependent PEX (osPEX). The goals and interactions of these three parts are described in this chapter. Details on the implementation of PEX-ME are given in the *PEX-SI Porting Guide*.

3.1.1. Goals and Constraints

The following are the design goals of PEX-ME:

- Provide an easy mechanism for building servers that support both PEX subsets.
- Use the same lookup table, name set, and font resource implementations for both PEX subsets.
- Use the same rendering pipeline for both PEX subsets.
- Provide an easy mechanism for porting the rendering pipeline.
- Separate X resource management from the resource implementations.
- Separate resource implementations from rendering.

- Provide an easy mechanism for porting resource implementations.
- Provide a well-defined interface between diPEX and ddPEX.
- Provide an extension that can be linked to the X sample server and run with minimal effort.

One constraint was defined to achieve the goal of ease of porting to existing X11R4 servers:

- PEX-ME must use the X11R4 ddx interface as its rendering foundation.

3.2. diPEX

diPEX provides the device-independent functionality of PEX-ME. The anticipation that support of the PEX protocol will be ported to native high-end graphics engines has lead to the design of a diPEX layer that is as thin as possible. It performs the following functions:

- Performs byte-swapping and floating-point conversions for requests and replies.
- Provides all X resource identifier to resource data de-referencing.
- Implements search context, pipeline context, and renderer resources.
- Isolates device-independent errors and generates error messages as necessary.
- Sends replies and error messages to the client.

diPEX receives PEX packets, unpacks them, and translates them into the correct (server-native) byte-ordered form and floating-point format. diPEX checks for errors and if none are found, dispatches the translated requests to the appropriate ddPEX module. In addition, diPEX handles replies to the client and performs resource management functions.

3.3. osPEX

osPEX provides the PEX-ME functionality that is heavily dependent on the target operating system. osPEX performs the following operating system-dependent functions:

- Locates a PEX font file in a specified directory.
- Loads a PEX font from a PEX font file.

Only 4.2 BSD implementations of these functions are provided in PEX-ME.

3.4. ddPEX

ddPEX provides the non-OS-related porting layers of PEX-ME. ddPEX is designed to provide portability to a wide range of graphics hardware, from simple frame buffers, to display-list devices, to advanced graphics engines. To achieve this, ddPEX performs the following functions:

- Renders individual structure elements and output commands.
- Controls the internal storage format of server-side structures.
- Edits the contents of server-side structures.
- Traverses the structures for rendering, picking, and spatial search.
- Performs hit-tests for picking and spatial search traversals.

- Maintains workstation and renderer state information.
- Maintains lookup tables and name sets.
- Loads PEX font definitions.

The support for these ddPEX functions is divided into five modules. Each module is designed to be as functionally independent of the other modules as possible. The modules are divided such that one module implements only those resources that support the PHIGS workstation subset; a second module implements those resources that are shared by both PEX subsets; a third module provides rendering control for both subsets; and two modules form the rendering pipeline for both subsets.

The five modules of ddPEX are the following:

- PHIGS Workstation Module
- Shared Resources Module
- Rendering Control Module
- Transformation Module
- 3D Rendering Module

The X11R4 ddx serves as the bottom rendering layer for PEX-ME.

The first two modules implement PEX resources. The PHIGS Workstation Module implements the PHIGS workstation, structure, and pick measure resources. It calls into the Transformation Module to reformat structure elements into the device-dependent format used during rendering. The PHIGS Workstation Module also provides a structure traverser that traverses structure elements. This structure traverser calls into the Rendering Control and Transformation Modules to realize the draw, pick, and incremental spatial search operations on the structure elements.

The Shared Resources Module implements the lookup table, name set, and PEX font resources. These resources are used in both PEX subsets. Most of the procedures in the PHIGS Workstation Module and the Shared Resources Module are called by diPEX.

The resources supported in the PHIGS Workstation and Shared Resources Modules are implemented as independently as possible, so that the support for a specific type of resource can be modified or replaced with minimal effect on the other resources. The PHIGS Workstation and Shared Resources Modules are discussed in further detail in Sections 3.6 and 3.7.

The Rendering Control Module provides control of the rendering pipeline. This includes initializing the device-dependent rendering state, tracking the rendering state, and processing output commands for immediate mode rendering. The Rendering Control Module initializes the rendering pipeline for the drawing, picking, or search operations. Most of the procedures in the Rendering Control Module are called by diPEX. Further details on the Rendering Control Module are given in Section 3.8.

The Transformation and 3D Rendering Modules are dedicated to supporting the rendering pipeline in PEX-ME. These modules are not exposed to diPEX. The Transformation Module reformats output commands and structure elements to an internal device-dependent rendering format. It updates the device-dependent rendering state according to the attribute-setting commands it receives. The Transformation Module also supports geometric transformations, and the lighting and clipping of the reformatted primitives. It passes the transformed primitives to the completely subordinate 3D Rendering Module, and performs hit-testing for picking and spatial search. Further details on the Transformation Module are described in Section 3.9.

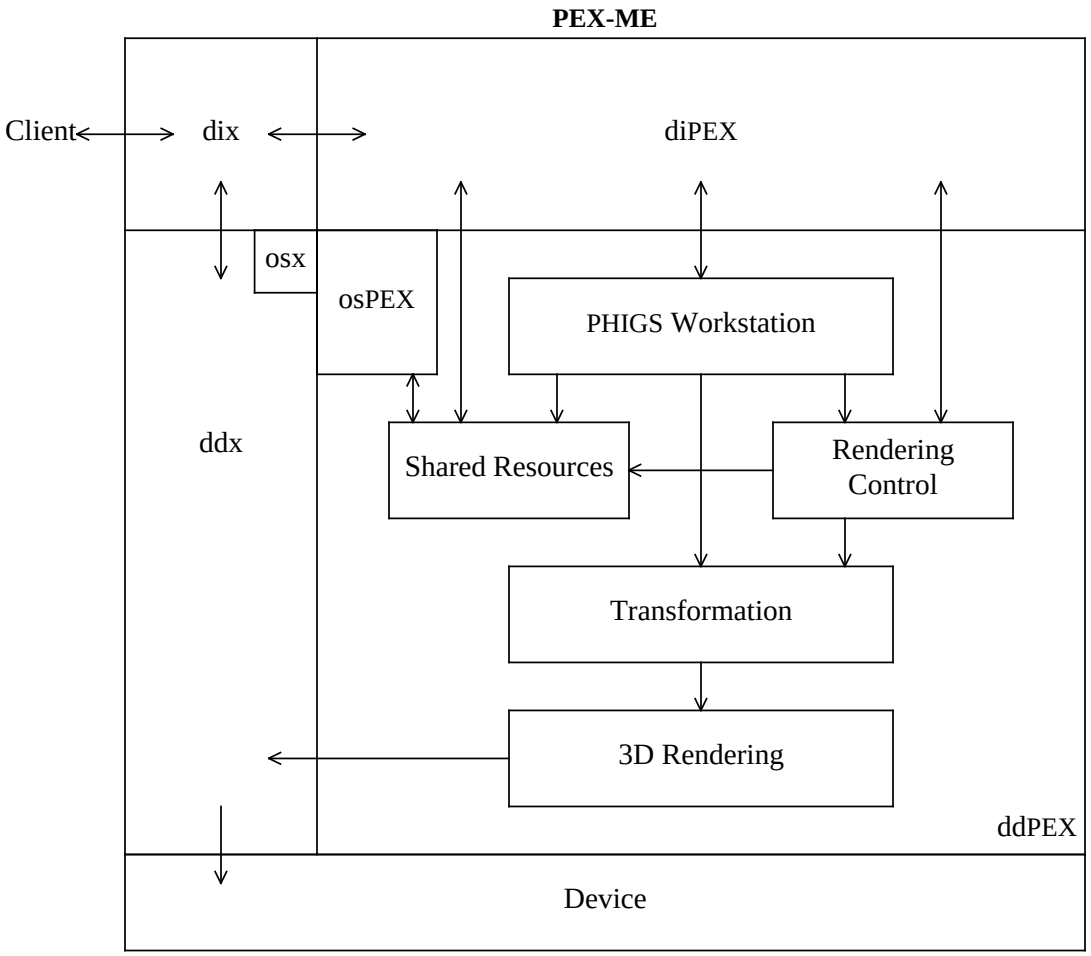
The 3D Rendering Module takes the transformed primitives from the Transformation Module and prepares the data for rendering by ddx. Further details on the 3D Rendering Module are given in Section 3.10.

Figure 3-1 shows the architecture of ddPEX and how it fits into the overall PEX-ME architecture. All client requests are received by dix, which examines and dispatches them appropriately. PEX requests are passed to diPEX.

diPEX unpacks the requests, and dispatches them to the appropriate ddPEX modules. diPEX calls into the PHIGS Workstation and Shared Resources Modules to dispatch requests that are part of the PHIGS workstation subset of PEX. The PHIGS Workstation Module subsequently calls into the Transformation Module to reformat and render output commands.

diPEX calls into the Shared Resources and Rendering Control Modules when dispatching requests that are part of the immediate mode rendering subset of PEX. The Rendering Control Module subsequently calls into the Transformation Module to reformat and render output commands.

Figure 3-1 *Server Architecture*



3.4.1. Communication Between diPEX and ddPEX

diPEX and ddPEX communicate through a stable, well-defined interface. The interface defines the ddPEX procedures called by diPEX. It is expected to be useful to any PEX-ME implementation and is not intended to be changed. A general description of the diPEX/ddPEX interface is given in the remaining sections of this chapter. The C binding and the PEX-SI implementation details are given in the *PEX-SI Porting Guide*.

The interfaces to the Transformation and 3D Rendering Modules are ddPEX-internal interfaces. These interfaces can be modified to support alternate data formats.

Because some of the replies generated by ddPEX are variable in length, diPEX cannot know how much space is needed for the reply data and thus cannot allocate the space specifically. To handle this, diPEX passes a buffer pointer to ddPEX. diPEX also passes the size of the buffer and the offset into the buffer where ddPEX should put the data. ddPEX places the data into the buffer at the specified offset and returns the buffer and an active data count. ddPEX is always responsible for properly formatting the data returned to diPEX to conform to the encoding requirements of the PEX protocol. With this scheme, diPEX can pass ddPEX the buffer that is used for the reply packet, and ddPEX can then fill in the reply data in the format expected by the client. This reduces the amount of data copied by diPEX when creating the reply packet.

A utility procedure is provided in diPEX for reallocating an existing buffer at a new size. ddPEX can use this procedure if the buffer is not large enough to hold the data to be returned.

3.5. PEX-ME Resource Support

PEX-ME fully supports the nine resource types defined for the PEX Protocol. They are:

PHIGS Workstation A large, complex resource that embodies much of PHIGS policy.

Search Context A resource used in performing searching operations on structure networks.

Pick Measure A resource used in conjunction with a PHIGS workstation resource to perform graphical selection of primitives.

Lookup Table Tables of values (of various categories).

Name Set Filters used for picking, highlighting, and visibility when rendering.

Structure Repositories of output commands, structures may be linked hierarchically into structure networks.

Renderer The basic drawing mechanism.

Pipeline Context A table of initial rendering attributes for a renderer.

PEX Font Stroke fonts and bitmap fonts.

Some of this resource support is implemented in diPEX. Table 3-1 shows which modules utilize which resources, and how.

Table 3-1 *Resource Types and Their Utilization*

Resource	*diPEX*	*PHIGS Workstation*	*Shared Resources*	*Rendering Control*
Search Context	M	A		
Pipeline Context	M			E
Renderer	M			A
PHIGS Workstation		M		
Structure		M		
Pick Measure		M		
PEX Font			M	
Lookup Table			M	
Name Set			M	

Legend: The table shows which modules affect or utilize which resources. The letters M, A, and E indicate that the module:

- M Manages or owns the resource.
- A Can alter the data in the resource.
- E Is exposed to the resource, but cannot manage or alter it.

3.5.1. diPEX Resource Management

diPEX handles all PEX resource ID to data de-referencing. All calls to the core X server routine *AddResource* are done in diPEX. Also, all PEX resource IDs in a PEX request are de-referenced by diPEX (via the core X server routine *LookupIDByType*) before diPEX calls ddPEX to handle the request.

The PEX resource support is divided into two groups based on which portion of PEX-ME manages (or ‘‘owns’’) the internals of the resource, diPEX or ddPEX. The resources that diPEX implements are called ‘‘visible resources.’’ The resources that ddPEX implements are called ‘‘semi-opaque resources.’’

The visible resources are:

- Renderers
- Pipeline Contexts
- Search Contexts

diPEX contains support for the creation, freeing, copying, modification, and inquiry of these resources. ddPEX uses these resources for other operations associated with them, and may modify the contents of renderers and search contexts.

The semi-opaque resources are the following:

- PHIGS Workstations
- Structures
- Pick Measures
- Lookup Tables
- Name Sets
- PEX Fonts

The semi-opaque resources have a portion that is visible to diPEX, but the bulk of their data is not utilized or managed by diPEX. The visible part of semi-opaque resources contains the PEX resource ID and possibly some other cached information, in order to simplify the interface between diPEX and ddPEX. For example, the visible part of a lookup table also includes the type of the lookup table.

3.5.2. Other diPEX Resources

diPEX also manages two internal resource types, *Context* and *WksDrawable*. These resources are used for diPEX internal bookkeeping and are not exposed through the protocol or to ddPEX.

The Context resource is associated with an individual client. An instance of this resource is used to keep pointers to the swapping and floating point conversion tables and routines that will be used to process requests from the client. An instance of a Context resource is created when the first PEX request is received from each PEX client. The resource is freed when diPEX receives notification of client shutdown.

The WksDrawable resource is used to detect deletion of a drawable resource that has been associated with a PHIGS workstation resource. This is important because if a drawable is deleted before its associated PHIGS workstation resource, via either direct client program control or during client cleanup in the server, the server needs to prevent any implicit redrawing of the workstation's image onto the defunct drawable. ddPEX is notified of any relevant drawable destructions.

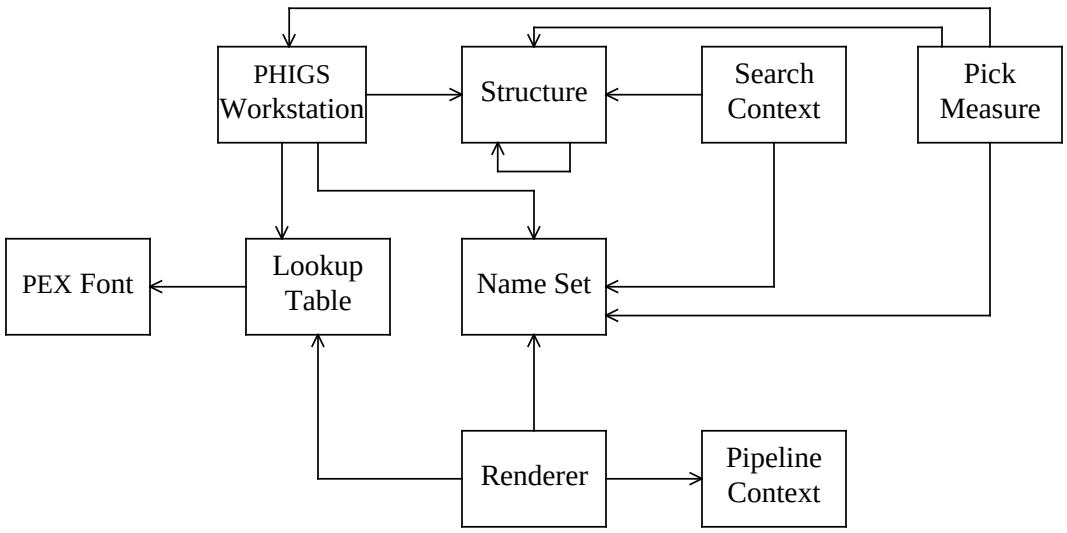
3.5.3. Resource Dependencies

PEX resources often depend on the existence of other, ''subordinate'' PEX resources. For instance, a PHIGS workstation resource contains references to the lookup table and name set resources. A lookup table resource that is referenced by a PHIGS workstation resource cannot be deleted until the workstation resource no longer needs the lookup table. These semantics are defined in the *PEX Protocol Specification* description for the *PEXFreeLookupTable* request and the other requests for freeing resources.

Resource dependencies such as these must be tracked by diPEX and ddPEX. The tracking of the resource dependencies is achieved by maintaining resource cross-references within each resource. The cross-references may be in the form of a full list or a simple reference count. Utility procedures that assist with the maintenance of the cross-references are defined for each resource type. These procedures add or remove a resource from the cross-reference list or count.

Resource dependencies in PEX-SI are shown in Figure 3-2. The arrow from resource type A to resource type B (A→B) indicates that resource type A depends on resource type B.

Figure 3-2 *PEX Resource Dependencies*



As an example, consider a colour lookup table resource named C-LUT. This resource has two cross-reference lists, one for PHIGS workstations and one for renderers. Now, consider that a PHIGS workstation resource named WKS-1 is created and the *PEXCreatePhigsWks* request specifies that the colour lookup table for this resource is C-LUT. The ddPEX procedure that creates the workstation resources calls the lookup table cross-reference utility to add WKS-1 to the PHIGS workstation cross-reference list of C-LUT. If a *PEXFreeLookupTable* request freeing C-LUT is then received, ddPEX checks the PHIGS workstation and renderer cross-reference lists in C-LUT to see if both lists are empty. C-LUT is immediately destroyed if the lists are empty. In this example, ddPEX sees that there is a workstation resource WKS-1 using C-LUT. Therefore, C-LUT is not immediately destroyed, but a flag is set in C-LUT indicating that it can be destroyed as soon as both of its cross-reference lists are empty.

When a *PEXFreePhigsWKS* request freeing WKS-1 is received, ddPEX calls the lookup table cross-reference utility again to remove WKS-1 from the PHIGS workstation list in C-LUT. The utility then checks for the flag that indicates if C-LUT has been freed. In this example, it finds the flag is set, and it finds that both cross-reference lists of C-LUT are empty. So, the utility destroys C-LUT.

In summary, a cross-reference *list* is only needed when a subordinate resource needs to notify its parent resource that a change has been made to the subordinate resource. The subordinate resource maintains a cross-reference *count* when no notification is needed. Further details on resource cross-references are given in the descriptions of each resource.

3.6. The PHIGS Workstation Module

The PHIGS Workstation Module implements the PHIGS workstation, pick measure, and structure resources. It also implements a structure traverser that passes structure elements to the Transformation Module for rendering, picking, or spatial search.

3.6.1. PHIGS Workstation Resources

A PHIGS workstation resource contains information on the state of the workstation's display, on the rendering state used during structure traversal, and on the supported pick devices. It updates the workstation display according to the current update mode and dynamics of the workstation.

Updating the display may require an implicit traversal of the structures posted to the workstation, as described by PHIGS and PHIGS PLUS. The rendering state information in the workstation resource is used by the rendering pipeline, which processes the structure elements traversed when the display is updated.

PEX-ME uses the same rendering pipeline to update the workstation display that it uses for the immediate mode rendering of output commands in a renderer resource. This rendering pipeline is designed to maintain its rendering state in a renderer resource. Thus, the PHIGS workstation resource provides a renderer resource to the rendering pipeline in PEX-SI. (For more information on the rendering pipeline, see Section 3.8 on the Rendering Control Module, and Section 3.9 on the Transformation Module.)

Other implementations may use a different rendering pipeline, a hardware pipeline for example, to update the display. The *PEX-SI Porting Guide* explains how the workstation resource can be modified to support a different rendering pipeline.

ddPEX defines the procedures that support the PHIGS workstation resources. It defines a set of procedures that support the PEX requests specified in the *PEX Protocol Specification*. ddPEX also defines utility procedures for updating the display and for updating the resource cross-references in the workstation resources. A general description of these procedures is given below. The procedures that support the requests for pick devices are described in Section 3.6.4 on pick measure resources.

The ddPEX procedures that support the PEX requests for PHIGS workstation resources map one-to-one with the requests specified in the *PEX Protocol Specification*. diPEX dispatches the requests to these procedures. The cross-reference update procedure adds and removes resources from the cross-references associated with workstation resources, as explained in Section 3.5.3.

The only resources that depend on workstation resources are pick measure resources. A workstation resource keeps a reference count of the number of pick measure resources that are using it. The update procedure is called by ddPEX to increment or decrement the count when a pick measure resource is created or destroyed. This procedure is not called by diPEX.

The dynamics utility procedures handle the display dynamics of workstation resources. These utilities are called by ddPEX when changes are made to the structure, name set, lookup table, and PHIGS workstation resources. The utilities determine how to update the workstation's display based on the workstation's dynamics and update mode. These utilities are not called by diPEX.

3.6.2. Structure Resources

Structure resources contain information on the structures that exist in the server-side CSS. The structure resources contain state information on these structures and on all of the elements in the structures.

ddPEX contains a set of procedures that support the PEX protocol requests defined for structure resources. It also has procedures that support the interaction of structure resources with other resources. PEX requests are handled by a set

of procedures that map one-to-one with the protocol requests specified in the *PEX Protocol Specification*, with the exception of the *PEXDestroyStructures* request. For this request, the ddPEX procedure destroys only one structure at a time. diPEX calls this procedure for each structure specified in the *PEXDestroyStructures* request.

Structure resources are part of a structure network. The network hierarchy is defined by the existence of *ExecuteStructure* elements in the structures. Structures maintain a cross-reference list of their parent and offspring structures in this structure network. This aids in maintaining the structure network and handling some of the protocol requests for structures.

Structure resources are subordinate to PHIGS workstation, search context, and pick measure resources. A cross-reference update procedure is provided for structure resources to maintain the cross-references of those resources that are using the structures. This utility procedure adds resources to and removes resources from the cross-reference lists and counts. It is called by both diPEX and ddPEX.

Structures are subordinate to the workstation resources on which they are posted. A structure resource maintains a cross-reference list of these workstation resources and notifies the workstation resource when the structure has changed, so that the Workstation can update the display. A structure resource maintains two cross-reference lists. One list contains the workstation resources that the structure is directly posted to. The other list contains the workstation resources that have an ancestor or descendent of the structure posted to them.

A structure resource keeps a reference count of the search context and pick measure resources that specify the structure in their start paths. A procedure is also provided that verifies that a path is valid within a given structure network. This procedure is called by both diPEX and ddPEX.

In addition to the diPEX/ddPEX interface just described, an internal interface handles operations on structure elements. This interface provides a mechanism that allows the storage format of structure elements to be defined by the implementation. The interface defines six operations on structure elements: create, destroy, copy, replace, inquire, and execute. There is a procedure to support each operation for every structure element type. The procedures are accessed through six procedure tables, each of which contains an entry for every table type, plus one additional entry to handle out-of-range elements.

The *createCSSElement* operation creates a new structure element from a given PEX element. This operation massages the data into the device-dependent format and returns a pointer to the new element. All data must be copied from the PEX request packet element to the structure elements because the request buffer is volatile.

The *destroyCSSElement* operation destroys the structure elements created by the *create* or *copy* procedures.

The *copyCSSElement* operation creates a new element by copying the data from the given element to the new element.

The *replaceCSSElement* operation replaces the given element with a new element. This operation is used only when an element is stored in replace mode and the element type being replaced is identical to the element being stored.

The *inquireCSSElement* operation converts the given element data to PEX Protocol format.

The *executeCSSElement* operation performs the action required for the given element during structure traversal. This could be rendering, picking, or spatial search. In PEX-ME, this operation is the Transformation Module's *ExecuteOC* operation (see Section 3.9).

The structure storage format used in PEX-ME is defined by the device-dependent format used in the Transformation Module. It is the same format that is used for rendering immediate mode output commands. The procedures in the CSS tables can call directly into the corresponding Transformation Module procedures. For more information on the Transformation Module, see Section 3.9.

3.6.3. Search Context Resources

Search context resources are defined and controlled by diPEX. diPEX handles all of the actions necessary to support the create, free, get, and change operations on Search contexts. ddPEX has one procedure defined to handle the *PEXSearchNetwork* request. This ddPEX procedure controls structure traversal for the spatial search. It turns on spatial search, traverses the structure network, and turns off spatial search. Turning on spatial search causes the rendering procedures in the *executeCSSElement* table to be replaced with procedures that do spatial search intersection, or hit testing. This procedure also sets the traverser to test for a hit after every graphics primitive is traversered. When a hit occurs, the traverser sets the search path in the search context resource and stops traversing.

3.6.4. Pick Measure Resources

Pick devices are supported as part of the ddPEX PHIGS workstation resources. They provide device-dependent initialization values for pick measure resources. ddPEX has procedures to handle the *PEXGetPickDevice* and *PEXChangePickDevice* requests. The procedures are called by diPEX when it dispatches these requests.

Pick measure resources are fully implemented in ddPEX. There is one ddPEX procedure to handle each pick measure protocol request. diPEX calls the procedures when it dispatches the requests.

The ddPEX procedure that handles the *PEXUpdatePickMeasure* request initiates structure traversal for the pick. It calls procedures in the Rendering Control Module to set up the graphics pipeline for picking before traversal, and to reset the pipeline when the traversal is finished. During traversal, the pipeline procedures for picking are called and the traverser checks for a hit and sets the pick path in the pick measure resource.

Hit testing is based on a ‘‘Pick First’’ algorithm.

3.7. The Shared Resources Module

The ddPEX Shared Resources Module supports the lookup table, name set, and PEX font resources. These resources are common to both the PHIGS workstation subset and the immediate mode rendering subset, and are used in all PEX server configurations (PHIGS workstation rendering, immediate mode rendering, or a combination of both).

The procedures for these functions map one-to-one with the PEX requests for the lookup table, name set, and PEX font resources. The procedures provide the functionality specified in the *PEX Protocol Specification*. The C binding and implementation details for these procedures are given in the *PEX-SI Porting Guide*. The Shared Resource Module interface is exposed to diPEX.

3.7.1. Lookup Table Resources

There are 12 types of lookup tables defined in the *PEX Protocol Specification*. Each type of lookup table contains information that is used during rendering. PEX-ME defines one ddPEX procedure for each lookup table request, regardless of the table type. All of these procedures are called by diPEX.

Lookup tables have cross-reference lists of the PHIGS workstation and renderer resources that use the tables. There is a utility procedure called by diPEX and ddPEX to update the lists. A utility procedure also checks that a lookup table can be used with a given drawable, based on the example drawable given when the lookup table was created. The procedure is called by diPEX and ddPEX when a lookup table is first referenced by a workstation or renderer resource, It checks to see if the table is compatible with the workstation’s or renderer’s drawable.

Each time a lookup table is changed, each workstation and renderer in the cross-reference lists is notified of the change through a call-back procedure defined for this purpose. The workstation call-back procedure updates the workstation state and the picture as needed. (For more information on the PHIGS Workstation Module, see Section 3.6.) The renderer call-back procedure sets the change flag in the renderer that is used when the renderer is validated. (For more information on the Rendering Control Module, see Section 3.8.)

3.7.2. Name Set Resources

Name set resources contain the lists that are used for highlighting, visibility, detectability filters, and the current name set. There is one procedure in ddPEX for each PEX name set request. These procedures are called by diPEX.

Name sets have a cross-reference list of workstation and renderer resources. When a change is made to a name set, each workstation and renderer in the cross-reference lists is notified of the change through a call-back procedure. The workstation call-back procedure updates the workstation state and the picture as needed. (For more information on the PHIGS Workstation Module, see Section 3.6.) The renderer call-back procedure sets the change flag in the renderer that is used when the renderer is validated. (For more information on the Rendering Control Module, see Section 3.8.)

3.7.3. PEX Font Resources

The PEX font resources contain the server native description of a PEX font. ddPEX procedures for font resources are found in the Shared Resources Module of ddPEX and in osPEX. These procedures support each of the PEX font requests.

Font resources have a cross-reference count of the lookup tables that reference the fonts. There is a utility procedure for updating the count that is called by the ddPEX lookup table code.

3.8. The Rendering Control Module

The Rendering Control Module controls the beginning of the graphics pipeline for the PHIGS workstation and the immediate mode subsets of PEX. This module maintains the device-dependent data in the renderer resources. It controls initializing, changing, saving, and restoring the device data during immediate mode rendering, workstation traversals, and during rendering, picking, and searching traversals. Each of the procedures defined for this are described in this section.

3.8.1. Pipeline Context Resources

A pipeline context resource is used to specify the initial attribute state when rendering begins. As stated in the *PEX Protocol Specification*, the attributes in a pipeline context resource are copied to the renderer resource at that time. No ddPEX procedures are associated with pipeline context resource requests.

3.8.2. Renderer Resources

The renderer resource contains device-independent and device-dependent information.

ddPEX manages the device-dependent information. This includes a procedure table for the rendering procedures, called the *ExecuteOC* table, and a device-dependent rendering state structure, called the *ddContext*. The *ExecuteOC* table is part of the Transformation Module interface and is described in Section 3.9. The *ddContext* is designed to be what its name implies — a structure that can be replaced or modified to interface with the software and hardware dependencies used in a particular PEX server implementation. *ddContext* is dependent on the specific devices used to render PEX primitives. *ddContext* contains state information, caches, and so on, in a form suitable to the native implementation. The attributes from the pipeline context resource and the device-independent part of the renderer resource are used to update the *ddContext*.

The Rendering Control Module contains procedures to create and destroy the *ddContext*. It also has procedures to support each of the following PEX requests:

- *PEXGetRendererDynamics*
- *PEXRenderOutputCommands*
- *PEXBeginRendering*
- *PEXEndRendering*
- *PEXBeginStructure*
- *PEXEndStructure*

These procedures have the required functionality for their matching PEX requests. All of these procedures are called by diPEX and by ddPEX.

The *BeginRendering* and *EndRendering* procedures establish the graphics pipeline as a rendering pipeline. The graphics pipeline produces graphical output to the drawable. In addition to these procedures, this module contains procedures for controlling the graphics pipeline for picking and for incremental spatial searches. Procedures for *BeginPicking*, *EndPicking*, *BeginSearching*, and *EndSearching* control those pipelines just as *BeginRendering* and *EndRendering* do for the rendering pipeline. These control procedures load the *ExecuteOC* table in the renderer with the correct procedures for picking or searching. In addition, there is a procedure for returning the status of the pick hit tests done in the lower level modules and an equivalent procedure for returning the status of the search hit tests.

The Rendering Control Module also has a utility procedure for validating the renderer. This procedure updates the *ddContext* with the current rendering attributes from the renderer and the pipeline context, lookup table, and name set resources it references. The validate procedure is called by the rendering control procedures for *BeginRendering*, *BeginPicking*, *BeginSearching*, and *RenderOutputCommand*s. It looks at a set of bit masks, which are in the renderer, to see which resources have been changed since the last time the renderer was validated. The procedure then updates the *ddContext* to reflect the changes according to the dynamics of the renderer. The bit masks are set by the call-back procedure, which notifies a renderer whenever one of its subordinate resources is changed.

3.9. The Transformation Module

The Transformation Module performs the following major functions:

- Provides the device-dependent CSS operations that create, destroy, copy, replace, and inquire structure elements for storage in device-dependent format.
- Reformats output commands into device-dependent data formats for immediate mode rendering.
- Processes device-dependent output commands (provided as structure elements or immediate mode output commands) for rendering.

The Transformation Module defines the procedures for transforming structure elements and output commands into a device-dependent format. It also defines other operations for manipulating structure elements and provides the first steps in processing the data through the graphics pipeline. It provides a different set of procedures, if needed, for rendering, picking, or searching operations.

Because structure elements are simply stored output commands, and structure elements are received by ddPEX in the same PEX protocol formats as output commands, structure elements and output commands are handled by ddPEX in the same way. Therefore, all references to output commands (OCs) in the rest of this document apply to both structure elements and to immediate mode output commands.

Six operations are defined in this module for manipulating OCs: create, destroy, copy, replace, inquire, and render. All of these operations are used for server-side CSS, as described in section 3.6.2. Only the parse, destroy, and execute operations are used for immediate mode output commands.

parse The parse operation reformats the PEX output command to the device-dependent OC format. This operation also allocates the storage needed for storing the device-dependent OC in the internal CSS.

destroy The destroy operation frees the storage allocated by the create operation.

copy The copy operation allocates storage for a new device-dependent OC and copies the data from a given device-dependent OC to the new one.

replace The replace operation copies the data from a PEX OC to a given device-dependent OC, thus replacing the contents of the device-dependent OC.

inquire The inquire operation returns the original PEX OC from a given device-dependent OC.

execute The execute operation starts the OC down the rendering pipeline. For OCs that set a rendering attribute, this operation updates the *ddContext* of the renderer. For output primitives, the execute operation performs geometric transformations, and clipping, and applies lighting attributes to the primitives. The primitives are then passed onto the 3D Rendering Module where the rest of the rendering pipeline is implemented.

Each of these operations is implemented separately for each output command type. The procedures are accessed through six procedure tables: *ParseOC*, *DestroyOC*, *CopyOC*, *ReplaceOC*, *InquireOC*, and *ExecuteOC*. Each procedure table contains an entry point for each output command type. PEX-SI uses procedure tables to provide flexibility for PEX-ME implementers. The default procedure tables supplied in PEX-ME can easily be modified to include implementation-specific procedures. Or the default procedures can be replaced for a subset of the output commands or all of the output commands. A single procedure can also be used for more than one output command.

The *ParseOC* table provides the procedures for creating the device-dependent output command from the PEX output command. The *DestroyOC* table provides the procedures for destroying the storage associated with output commands. The *CopyOC*, *ReplaceOC*, and *InquireOC* tables provide the procedures for copying, replacing, and inquiring the output commands, respectively. A table is defined in ddPEX for each of these five operations.

The *ExecuteOC* table provides the procedures for rendering the output commands. One *ExecuteOC* table is associated with each renderer resource (see Section 3.8.2). This allows for different rendering procedures to be used depending on the current rendering state. For instance, separate procedures could be provided for rendering hollow fill areas and filled fill areas. To do this, the procedures must be associated with the renderer resource that contains the rendering state. PEX-ME uses the *ExecuteOC* table to load different procedures for drawing, picking, and incremental spatial searches of primitives.

3.10. The 3D Rendering Module

The 3D Rendering Module is the rendering layer for ddPEX. The data received by this layer are either 3D or 2D device coordinates with model/view, transformation/clipping, lighting, and/or depth-cueing already applied. Conceptually, this layer resembles an extended ddx. In the case where the data has been resolved to 2D coordinates with no other per-vertex data, the input data can be passed directly to ddx. However, when interpolation is required for 3D vertices with per-vertex colour information, and so on, significant data manipulation is necessary.

Alternatively, when the graphics pipeline is set up for picking or searching, the 3D Rendering Module does the hit test for these functions. It does not call ddx to do this.

This module could be replaced completely if an application requires the data to be passed to implementation-specific software or hardware instead of ddx.

3.11. Miscellaneous ddPEX Functions and Errors

Four functions are defined to handle miscellaneous operations needed in all PEX implementations:

- Two functions support the *PEXGetEnumeratedTypeInfo* and *PEXGetImpDepConstants* requests. They return the implementation information for enumerated types and implementation-dependent constants.
- A third function initializes ddPEX whenever the server is started or reset.
- The fourth function is called when the server shuts down. It cleans up after ddPEX.

3.11.1. Errors in ddPEX

ddPEX returns an error code to diPEX for each ddPEX procedure performed. The error returned is the X- or PEX-defined error number specified in the PEX protocol for the particular condition.