UIPEX: DESIGN OF THE APPLICATION PROGRAMMER INTERFACE

BY

PARRIS KARL EGBERT

B.S., Utah State University, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

CHAPTER 1.

Introduction

In the past few years, the desire for easy-to-use, efficient, realistic computer graphics has steadily been on the increase. Computer graphics has found wide acceptance in engineering, architecture, space exploration, medicine, motion picture production, and many other diverse areas. Computers allow us to visualize things that are impossible or very difficult to view naturally. For example, an airline pilot can use a flight simulator to train with, and be presented with situations that he may never face in real life, or that would be too costly to perform in real life. As computers continue to increase in capability and decrease in size and cost, the need for high speed, realistic computer graphics will increase.

Several attempts have been made at standardizing a computer graphics interface language. The most notable of these are the Graphical Kernal System (GKS) [1], the Programmer's Hierarchical Interactive Graphics System (PHIGS) [2, 3], and the Programmer's Hierarchical Interactive Graphics System Plus Lumîere und Surfaces (PHIGS PLUS) [4, 6].

Another major event in making computers more user friendly was the introduction of windowing systems. A windowing system allows a user to have several *windows* available to run various jobs in simultaneously. This allows the user to run several processes at the same time, and thus increases the productivity of the user.

One of the most popular windowing systems today is the X Window System (X) [13]. Because of its ease of use and widespread availability, it has become the de facto industry standard.

In an attempt to incorporate PHIGS into X, the PHIGS Extensions to X (PEX) project was started at Digital Equipment Corporation [9, 11]. PEX extends the functionality of X to include the

capabilities of PHIGS and PHIGS PLUS, including 3D output primitives, light source shading, etc.

This thesis discusses the implementation of UIPEX - The University of Illinois implementation of PEX. Chapter 2 gives an overview of PHIGS and PEX. One portion of PEX is the Application Programmer Interface level or API. Chapter 3 discusses the API level, how it bridges the gap between the application programmer and PEX, and the functionality that exists at this level. Chapters 4, 5, and 6 discuss pertinent issues that arose in the API implementation. Several possible solutions are presented to the problems, as well as an explanation of why we chose the alternatives we did. The final two chapters discuss the strengths and weaknesses of PHIGS as a graphics standard, and present solutions for overcoming some of the weaknesses.

CHAPTER 2.

System Overview

2.1. PHIGS

The Programmer's Hierarchical Interactive Graphics System (PHIGS) was designed as an interactive 3D graphics standard in an attempt to provide a simple, yet powerful, computer graphics standard [3]. Many of the ideas behind PHIGS stem from the Graphical Kernal System (GKS) [1]. However, PHIGS has been extended to provide several capabilities that GKS does not. Some of the more notable of these are 3D output primitives, a hierarchical modeling scheme, and editable structures. These capabilities provide the application with a much more powerful graphics system than is present in GKS. In addition, PHIGS is currently being extended to include things such as more complex output primitives, depth cueing, and light source shading. This work, which is currently a proposed ANSI standard, is known as PHIGS PLUS [4].

2.1.1. PHIGS Structures

In PHIGS, objects are stored as *structures*. A *structure* is comprised of a list of *structure elements*. A *structure element* can be an output primitive, an output primitive attribute (such as line color), a geometric transformation, a label to be used in structure editing, or a reference to another structure. All of the structures in PHIGS are maintained in the *centralized structure store* (CSS). The CSS keeps track of all of the information in the structures and performs updates on the structures when editing operations are performed by the application.

2.1.1.1. Structure Networks

As mentioned above, structures may reference other structures, via the EXECUTE STRUCTURE structure element.

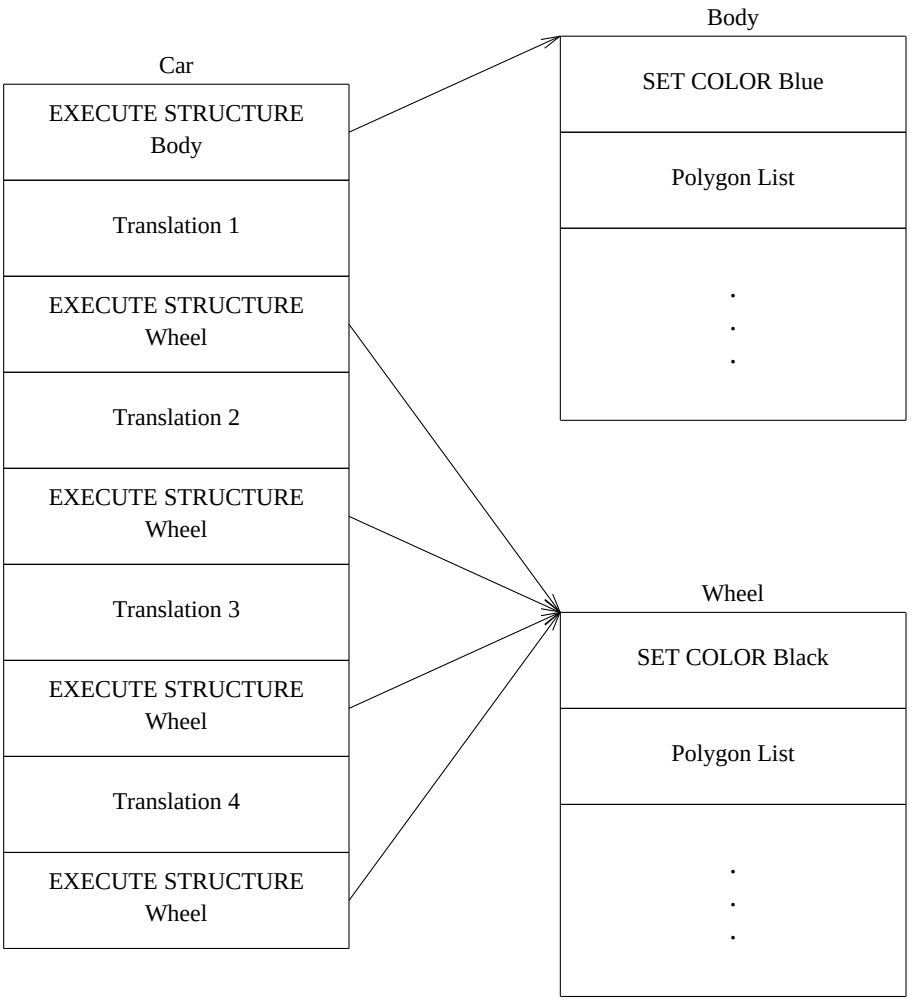

Figure 1. Modeling a Car Using the PHIGS Structure Network Capabilities.

This feature allows the programmer to create objects in a hierarchical fashion and make multiple instantiations of objects. Figure 1 shows how this functionality could be used in the creation of a car.

2.1.1.2. Posting Structures

Once a structure has been created, PHIGS allows the user to perform a variety of actions on that structure. To view the structure, the POST STRUCTURE command is invoked. This command sends the structure to the rendering pipeline so that is can be viewed on a workstation. The structure itself remains in the CSS unchanged. Likewise, an UNPOST STRUCTURE command will remove the structure from the workstation it is being viewed on, but will leave it unchanged in the CSS.

2.1.1.3. Structure Editing

Structures can be altered by adding new structure elements, changing existing structure elements, or deleting structure elements. Only the currently open structure can be edited. There is a structure element pointer associated with the current open structure that determines where the next structure element is to be placed. The system can be in one of two edit modes, INSERT or REPLACE. If the current edit mode is INSERT, new structure elements are inserted into the structure immediately following the structure element pointer, and then the pointer is advanced to the next structure element. Structure elements that are already in the structure remain intact.

If the edit mode is REPLACE, a new structure element will be inserted into the structure at the position pointed to by the structure element pointer. The structure element currently there will be destroyed. After replacing the structure element, the pointer then points at the currently placed element.

2.1.1.4. Archiving Structures

PHIGS also allows structures to be stored in archive files so that they can be used in the future. Structures that have been created in PHIGS can be stored in an archive file by issuing an ARCHIVE STRUCTURES command. Structures that have previously been saved in an archive file can be

loaded back into PHIGS via the RETRIEVE STRUCTURES command. This provides an application a method of long term storage for structures that have been created.

2.1.2. Workstations

A PHIGS *workstation* is an abstract PHIGS entity that is used, among other things, for the displaying of PHIGS structures. A workstation can accept input, output, or both.

There can be many workstations open at any given time in PHIGS. Each workstation has a type associated with it. The workstation types are *output, input, input/output, metafile input,* and *metafile output*. *Output* workstations are capable of displaying PHIGS output primitives, but cannot perform input operations. A workstation of type *input* has at least one input device associated with it and is capable of performing input operations but cannot display output primitives. An *input/output* workstation can display output primitives and can perform input operations. *Metafile input* workstations are used to retrieve data that has been stored in metafiles, and *metafile output* workstations are used to store data to metafiles.

Each open workstation has a workstation description table associated with it. This table keeps track of information about the workstation so that an application can query the workstation to determine the workstation's capabilities.

2.2. PEX

In the past few years, windowing systems have grown immensely in their usage and popularity. Programmers often desire to perform several tasks simultaneously, and a windowing environment gives them this capability. With the increase in speed of computers today, as well as the increase in the amount of memory available, windowing systems are becoming much more available and widespread.

One of the most popular windowing systems in use today is the X Window System (X) [13]. X was designed as a network transparent windowing system that provides windowing, input, and 2D output operations. Because of X's wide availability, and its ability to work in a heterogeneous workstation environment, it has become a de facto industry standard. However, X still has its shortcomings. At the time X was designed, the design team was specifically addressing the problem of a network transparent windowing system. Other problems, such as 3D graphics, were not part of the original X design goals. With the increase in computing power today, as well as the increase in the desire for realistic computer graphics, a task force was formed in 1986 at Digital Equipment Corporation to address some of these shortcomings. The work of this task force, combined with work from Sun Microsystems, eventually became known as X3D-PEX, or PEX [5, 11].

The main design goals of PEX were to:

1. Extend X to support 3D graphics.
2. Support a wide range of X platforms.
3. Provide efficient support for PHIGS and PHIGS PLUS.
4. Establish the PEX definition in a timely fashion.
5. Produce an extension that would be widely accepted by the X community.

X (and subsequently PEX) was designed in a client-server fashion. This makes it possible for several clients to run simultaneously on different machines, each accessing the server machine. PEX was designed by extending the functionality of the server, and adding additional messages to the protocol, so that the client can access that functionality [10, 12]. Figure 2 shows the PEX system model.

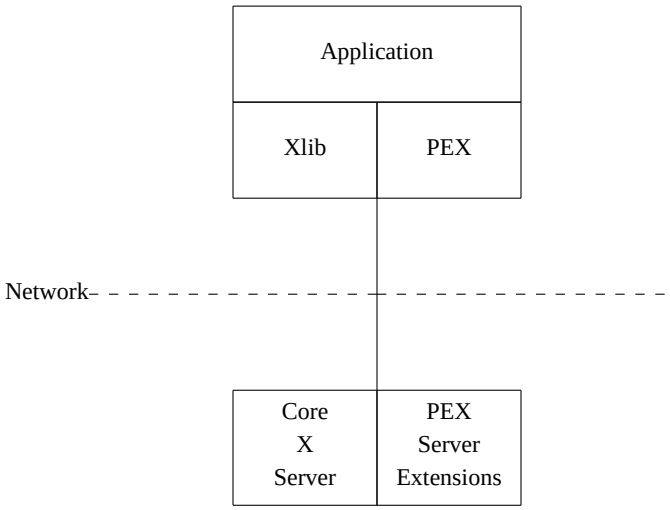

Figure 2. PEX Model.

When a client sends a protocol message to the server, the core X server checks the packet to see if it is an X message. If so, it services that packet. If it is a message that the core X server does not understand, it passes the packet to the PEX portion of the server and allows PEX to service the packet.

This model can be broken down into finer granularity, as shown in Figure 3. In this figure, the Application Programmer Interface (API), Modeling Extension (ME), Device Independent PEX (DIPEX), and Device Dependent PEX (DDPEX) levels are introduced.

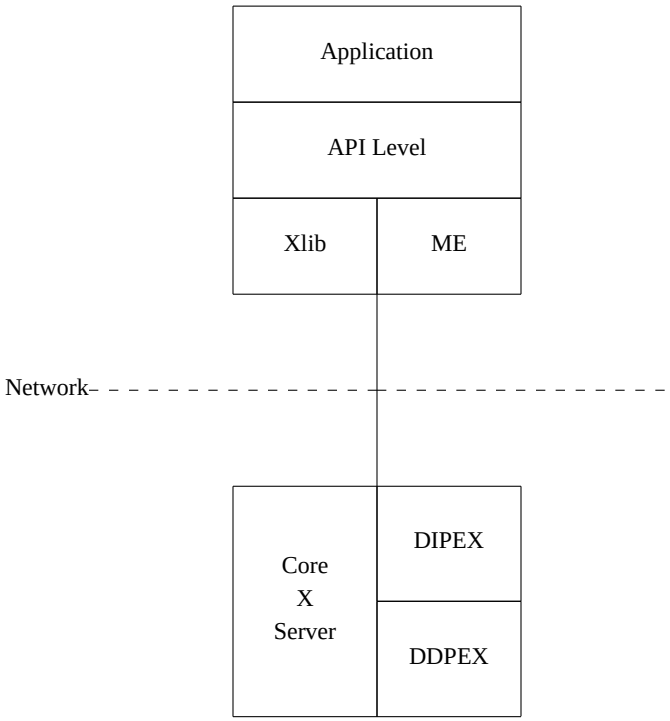

Figure 3. Extended PEX Model.

A PHIGS application talks to the API level. The API level in turn passes data and requests to the server via the ME level. The ME level takes care of network packet encoding, and sends the data across the network. The DIPEX level [14] receives PEX packets from the network, decodes them, stores the appropriate data, and then sends data to the DDPEX level for rendering. The DDPEX level then performs the rendering of the scene.

CHAPTER 3.

The Application Programmer Interface Level

The *application programmer interface* (API) level of PEX is the level that lies between the PHIGS application and the PEX server itself. The main functionalities of the API level are:

1. Receive user data, manipulate it into PEX format, and pass it to the server.
2. Retrieve data from the server, transform the data into PHIGS format, and return the data to the user upon the user's request to do so.
3. Perform error checking on user data, ensuring that the data is reasonable (where possible) and that the current state of PHIGS is consistent with the user's current request.
4. Provide error handling capabilities to be used when errors are detected.
5. Maintain the PHIGS system state, the state of all current PHIGS workstations, and the state of current PHIGS structures.
6. Provide input capabilities to the user.
7. Maintain PHIGS archive and metafiles, and provide a conversion from PHIGS and PEX data formats to the archive and metafile data formats.

The following sections describe these operations in more detail. In addition, [7] contains additional details on the actual implementation of the API level.

3.1. Manipulating PHIGS Data

The data types used in PHIGS are different than those used by the PEX server. Thus, a conversion must take place in order to pass PHIGS data to the server. This is done in each PHIGS function call that requires it. For example, in PHIGS, an element of a pick path (used in *pick* input operations) is stored in the data structure of type Ppick_path_el, which is defined as:

```
typedef struct {
        Pint  struct_id;  /* structure identifier              */
        Pint  pick_id;          /* hierarchical pick identifier     */
        Pint  el_num;           /* element sequence number          */
} Ppick_path_el;
```

The PEX structure that corresponds to this is a data structure of type PEXPickElementRef, which is defined as:

```
typedef struct {
        Structure   sid;  /* Structure identifier             */
        int         offset;  /* Offset into the structure     */
        int         pickid;  /* Pick identifier               */
} PEXPickElementRef;
```

Thus, when a PHIGS function call is made in which Ppick_path_el data is passed to it, The PEX structure identifier that is associated with this PHIGS structure identifier is found and stored in the *sid* spot in the PEX data structure. Then the PHIGS pick identifier and element number are stored in the *pickid* and *offset* spots, respectively.

3.2. Retrieving Data from the Server

PHIGS allows the user to request various information from the system. The types of information that can be requested are operating state values, PHIGS state values, workstation state values, structure state values, structure content values, and error state values. This process is the

reverse of that explained in the preceding section. API makes a function call to the server requesting the appropriate information. When the information is returned from the server, it must then be transformed into data that the PHIGS application can understand. Once this has been done, the transformed data is then returned to the user.

3.3. Error Checking

The PHIGS and PHIGS PLUS Functional Specifications [3, 4] outline the specific errors that must be checked for each PHIGS function. In addition, the PHIGS binding [2] outlines language specific errors that may also be encountered (mainly having to do with memory allocation). The first thing done in an API function is to check the error conditions stated in the Functional Specification. These are checked in the following order:

1. The PHIGS state, workstation state, structure state, and archive state are checked to make sure they are reasonable for the function that has just been called. For example, if the function is supposed to store a structure element to a structure, there must be an open structure. If not, an error is reported.

2. The values of input parameters are checked to make sure they are valid. This is not possible to check in all cases, but where there are restrictions on the input parameter values, the error checking ensures that those restrictions are adhered to.

3. System error conditions are checked for. For example, if a function attempts to allocate more memory than is currently available, this error will be reported.

3.4. Error Handling

If an error condition is found to exist, one of two actions are taken, depending on the type of function that was being executed when the error was detected. If the function was a PHIGS inquiry function, the type of error that was detected is recorded in the *error indicator* parameter that is passed to every inquiry function, and then a return is made from the function. If the function is not an inquiry function, the following actions are taken. First, the system error state is set to ERROR ON. Next, API makes a call to the error *handling routine*, passing the appropriate parameters. When execution returns from the error handling routine, the system error state is set to ERROR OFF. Finally, a return is made from this function call.

Because error checking is done many times, the error checking operation is done in two macros -- one for inquiry functions and one for non-inquiry functions.

The error handling function is designed so that an application can either use the system supplied function or provide its own. The error handling function takes as input parameters the error condition that was detected, the PHIGS function that detected the error, and the error file. The system supplied error handling function simply makes a call to the *error logging* function that is also supplied by PHIGS. The application is free to use its own error handling function and have it perform whatever operations it deems necessary.

The PHIGS error logging function has the same parameters as the error handling function. When a call is made to this function, it writes a message to the error file indicating the error condition that was detected and the PHIGS function that detected it.

3.5. Maintaining State Information

In PHIGS, state information must be maintained for a variety of entities. These include the current operating state, PHIGS traversal state, PHIGS state, workstation state, structure state, and error state. Of these, the operating state, portions of the workstation state, and error state values are stored in the API level. The others are stored in the server.

3.5.1. Operating State

The operating state is comprised of four components -- the system state, workstation state, structure state, and archive state. The system state is an indication of the state of PHIGS. It is set to PHIGS OPEN when the user makes a call to the *popen_phigs()* function call, and is set to PHIGS CLOSED upon the completion of a *pclose_phigs()* function call.

The workstation state value specifies the current state of workstations in PHIGS. If the value of this variable is WORKSTATION OPEN, then one or more PHIGS workstations are currently open. If it is set to WORKSTATION CLOSED, then there are no workstations currently open.

The structure state value specifies the current state of structures in PHIGS. If there is currently an open structure, the value of this variable is set to STRUCTURE OPEN. Otherwise it is set to STRUCTURE CLOSED.

The archive state value keeps track of the current state of archive files in PHIGS. If there is currently one or more open archive files, the value of this variable is set to ARCHIVE OPEN. Otherwise, it is set to ARCHIVE CLOSED.

3.5.2. Workstation State

Each open workstation in PHIGS has a workstation state list associated with it. This state list includes information about the type of workstation it is, the corresponding PEX workstation id, the list of input devices associated with the workstation, and other state information that pertains to the workstation. Much of this information is filled in when the workstation is opened by the PHIGS application. State information that is not available at that time is filled in as the user performs various actions (e.g., creating a new input device, etc.).

3.5.3. Error State

The PHIGS error state list consists of three components. These are the current error state of the system, the current error handling mode, and the current error file. The error state is set to ERROR ON when an error is detected in the system, and set to ERROR OFF when error handling has been completed.

The error handling state value specifies what the system is to do when an error is detected. If the value of this variable is ERROR HANDLING ON, the system makes a call to the error handling function. If it is set to ERROR HANDLING OFF, that function is not called when an error is detected.

The error file specifies the file to write error information to when an error is detected. The value of this state variable is set when the user makes a call to the *popen_phigs()* function call.

3.6. Input

All input, except some of the picking functionality, is performed at the API level. This includes initializing the input devices, receiving and interpreting input events, setting up the proper input

modes, etc. Chapter 6 discusses PHIGS input in detail.

CHAPTER 4.

Multiple Server Architecture

PEX was designed to allow an application the ability of connecting with multiple servers. To facilitate this capability, the question of where structure data is to be stored has to be addressed. There are four possibilities.

The first is to store the structures on the client side, and perform client side traversal on the structures. This provides a centralized storage location and ensures data consistency, since there are not multiple copies of the data being stored. There are two major disadvantages to this scheme. First, whenever a structure is posted to a workstation, the structure must be traversed and the entire resultant image sent across the network to the posting workstation. This can generate a high amount of network traffic. The second problem with this approach is that by performing client side traversal, the underlying hardware of the respective servers cannot be fully utilized. If a server provides hardware to perform fast rendering of the image, the client has no way of accessing that hardware.

The second alternative is to store all of the structures on one server and then simply send structures from this default server to other servers on an as-needed basis. Using this scheme, the default server always has the most up to date version of the structures. When a structure is posted to a different server, the default server sends the structure to the other server. When an editing operation is performed on a structure, all of the servers to which that structure is posted must be informed of that update by the default server. This method also guarantees data consistency, since the default server always has the correct data, and always communicates any changes to the servers that are affected. The disadvantage of this technique is that it has the potential of generating very large amounts of network traffic, since structures may need to be sent to servers multiple times.

The third possibility is to store all structures on all servers. When a structure is subsequently posted to a particular server, it already has all of the information it needs; it can simply post the structure from its storage. The major disadvantages to this scheme are data consistency problems and wasted memory. We must ensure that the data is kept consistent across servers. This means that anytime a structure editing operation is performed, all servers must be informed of that operation. In addition, each time a new display is opened, all of the structures currently in the system must be sent to it. Since all servers will store all structures, there may be several structures stored at a server that are never used by that server. This takes up unnecessary memory on that server. The advantage to this scheme is that structures only need to be sent to a server one time. This can substantially reduce the amount of network traffic generated.

The fourth possibility is to store each structure once in the system, but store different structures on different servers. This scheme works well if we know that each structure will only be posted on one server. However, the problems associated with keeping track of where a structure is stored, as well as those of sending structures between servers when needed, make the complexity of this scheme far outweigh its advantages.

The following analysis compares these alternatives. Let

s	be the number of servers in the system,
p	be the average number of posted structures per server,
u	be the average number of updates per server,
e	be the average number of elements per structure,
t	be the total number of structures,
i	be the average size of a generated image,
P	be the probability of having a particular structure stored at the server to which the structure is posted, and

o be the overhead associated with keeping track of where each structure is stored.

Then, for alternative 1 above, there are *s* servers that are updated *u* times, each of which are sent data of size *i*. Thus, the amount of network traffic is:

$$sui.$$

For alternative 2, all structures are sent to the default server, and each of the other (s - 1) servers are sent p structures of size e, and they are sent u times. Then the total amount of network traffic generated by this scheme is:

$$te + (s - 1)peu.$$

For alternative 3, each of the s servers is sent t structures, each of size e. Thus, the total amount of network traffic is:

$$ste.$$

For alternative 4, each structure is initially sent to one server, generating te amount of data. After this initial storage, each of the s servers is updated u times. Only structures that are not stored at a particular server need to be sent to it at update time. Thus, there are $(1 - P)p$ structures sent to each server at each update, and each structure is of size e. Thus, the total traffic is:

$$te + supe(1 - P) + o.$$

In alternative 1, the network traffic depends largely on the size of the image sent. It is assumed that images will be quite large, and thus the amount of network traffic will also be large. However, the

major drawback of this approach is that the rendering process cannot take advantage of the server's rendering hardware. This will greatly increase the amount of time needed to render an image. This attribute makes this alternative very undesirable.

Alternative 4 will work well when structures are only posted to a single server, and are stored at that server. In this case, $P = 1$, and the only network traffic is in sending a structure to its server at structure creation time. However, since structures can be posted on multiple workstations, this ideal situation may not exist. In addition, the major disadvantage of this scheme is the complexity involved in keeping track of where structures are located. When a structure is to be posted on a server other than the server it resides on, the server to be posted to must determine where the structure is located, and then request that the server that stores the structure send that structure to it. The complexity involved in this scheme outweighs the benefits derived from it. Thus, it appears that alternatives 2 and 3 are the only viable options.

The following analysis and derivation show the condition necessary for the amount of network traffic using alternative 3 to be less than that for alternative 2:

$$ste < te + (s - 1)peu$$

$$\rightarrow \quad (s - 1)te + te < te + (s - 1)peu$$

$$\rightarrow \quad (s - 1)te < (s - 1)peu$$

$$\rightarrow \quad t < pu \qquad \text{for } s > 1,\ e > 0.$$

Thus, the network traffic for alternative 3 will be less than that for alternative 2 when the number of structures in the system is less than the product of the average number of posted structures and the average number of updates per server. Assuming that most applications will have frequent changes made to them, either by changing the output primitives, changing the viewing position, or by other changes to the structure, then alternative 3 seems to be the best alternative of the four.

CHAPTER 5.

Structure Mapping

One of the differences between the PHIGS and PEX methodologies is the way in which structures are handled. In PHIGS, the user opens a structure and then can store structure elements in the structure. If he desires to store structure elements in a different structure, he must close the currently open structure and then open the new one.

In PEX, a structure is an X11 resource. All structures are open at all times. An application using PEX is free to store output commands to any structure that has previously been created.

To bridge this gap, API is required to remember which PHIGS structures are currently in the system, which PEX structures they correspond to, and which PHIGS structure is currently open (if any). Because multiple servers are supported in PEX, and because our implementation stores all structures at all servers, this complicates things even more. This chapter contains a discussion of the manner in which the PHIGS/PEX structure mapping is implemented.

Since PHIGS structure ids are simple integers and PEX structure ids are actual resource ids, there must be some mechanism of mapping between the two. Since there is no restriction on the PHIGS structure id number, the technique we chose in our implementation is to store the mapping in a hash table that is indexed by the PHIGS structure id. Each entry in the hash table is a pointer to a list of PHIGS/PEX structure id maps. The objects in the list are composite data items consisting of three fields. These fields are the PHIGS structure id, a pointer to a structure containing the mapping to PEX structure ids, and a pointer to the next item in the list. The middle item is the item of interest here. It is a pointer to a data structure that is also comprised of three fields. The first field is an integer that states the number of elements the remaining two fields will have.

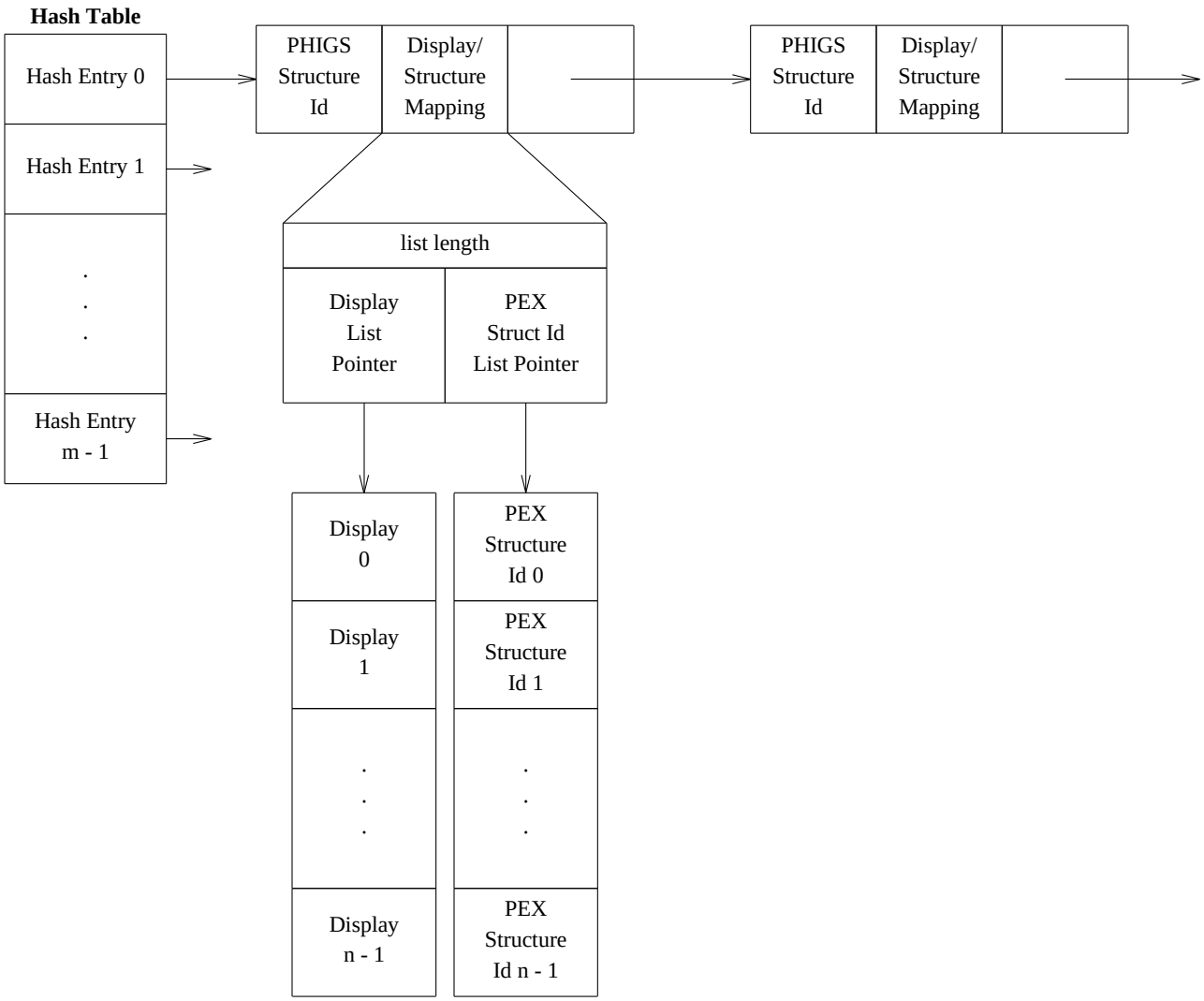

Figure 4. Organization of PHIGS/PEX Structure Mapping Table.

The second field is a list of all of the displays that are currently in use by this PHIGS application. The third field is a pointer to a list of PEX structure ids. These structure ids correspond to the PEX structure resources allocated on the corresponding displays for this particular PHIGS structure. Figure 4 illustrates the PHIGS/PEX structure map implementation of UIPEX, and Figure 5 gives a specific example.

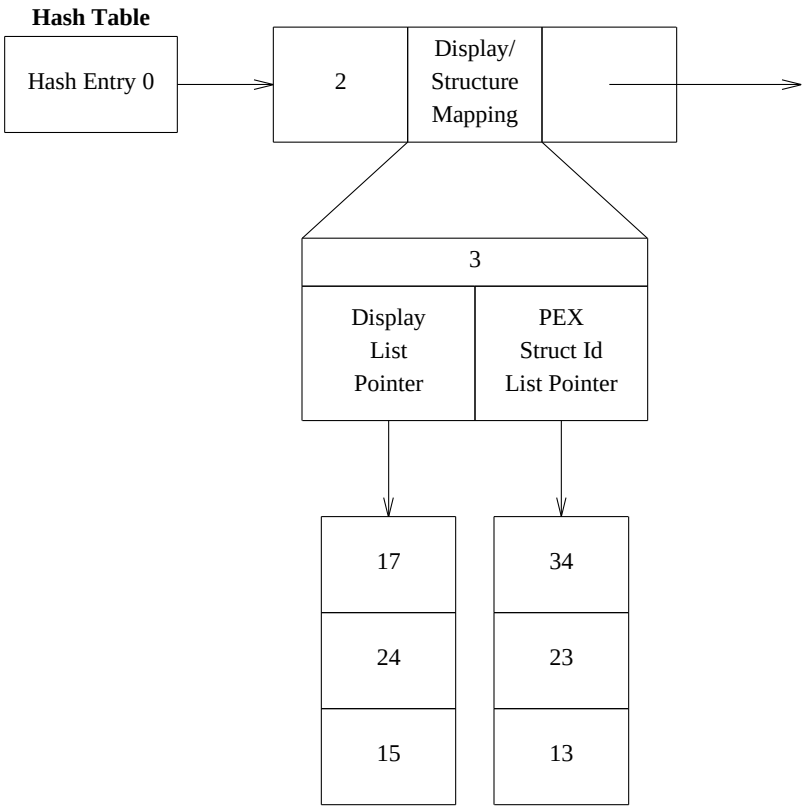

Figure 5. Specific Example of PHIGS/PEX Structure Mapping.

In the example of Figure 5, PHIGS structure number 2 hashes to hash table entry number 0. There are currently three displays in the system, numbered 17, 24, and 15. Since this PHIGS structure is stored at each of the displays, the corresponding PEX structure id is stored for each display. These are 34, 23, and 13. Thus, PHIGS structure 2 corresponds to PEX structure 34 on display 17, to PEX structure 23 on display 24, and to PEX structure 13 on display 15.

5.1. Opening a New Structure

When a new structure is opened in PHIGS, a new entry must be made in the PHIGS/PEX structure mapping table. The PHIGS structure id number is hashed to get the appropriate location in the hash table to go to. In the PHIGS system state list, a list of all currently known servers is kept.

Using this list, a new PEX structure is created on each of the servers. The PHIGS structure id, server list, and a list of the PEX structure ids that were just created are then placed in the structure mapping table.

5.2. Opening a New Display

Opening a new display requires more work. First of all, every PHIGS structure that is in the system must be copied to the new server. This is done by going through all entries in the structure mapping table and accumulating the lengths of their lists. This tells us the number of PHIGS structures currently in the system. Then we create that number of PEX structures on the new server and associate one PEX structure with each PHIGS structure. This association is done by going back through all entries in the structure mapping table and adding the new display and structure number. Then all of the structures are copied to the new display.

5.3. Reasons Behind This Design

There are at least two other plausible ways of designing the structure mapping table. One is to just keep one list of displays in the system and then eliminate the display list from each of the structure mapping entries. The other is to combine the display list and the PEX structure id list into one list, where entries in the list would be a (display, PEX structure id) pair. The first alternative was not selected because of the problems associated with keeping the various lists in the same order. If that approach had been implemented, we would have had to ensure that the ordering of the PEX structure id lists would be identical for every PHIGS entry. While this wouldn't have been extremely difficult to implement, the flexibility of the chosen design seemed to be more appealing.

The second alternative presented above was not chosen because of the way the API/ME interface is set up. In some of the ME function calls that are invoked, a parameter that is expected is

a list of PEX structure ids. If the structure mapping table had been implemented using a unified display/structure list, then the structure list would have to be extracted out of it, which would have added a needless step. To avoid that step, the data structure is left as two separate lists, and then a pointer to the PEX structure list is taken and passed directly to the ME functions.

5.4. Operations on the Structure Mapping Table

This section explains the operations that can be performed on the structure mapping table. These include actions such as storing information to the table, requesting information from the table, initializing the table, etc.

1. PhigsHashInit(). This function initializes the hash table by nullifying all of the entries in it. It is called when the user makes a call to *popen_phigs()*.

2. StoreStructMap(struct_id, pex_struct_id, display). This function takes as its arguments a PHIGS structure id, a PEX structure id, and a display. It then adds the mapping between these three into the table. This function is called when a new workstation is opened.

3. StoreMStructMap(struct_id, display_structure_list). This function is the multiple server equivalent of the preceding function. It takes as its arguments a PHIGS structure id and a list of (display, PEX structure id) pairs. It then adds this list into the correct spot in the table. This is called when a new structure is created or a PHIGS structure id is changed.

4. GetStructMap(struct_id, display). This function takes as parameters a PHIGS structure id and a display. It then searches the table to find this PHIGS structure. Once it has found it, it searches the corresponding (display, PEX structure id) pairs to find the given display. When it finds that display, it returns the corresponding PEX structure id. This is called by many API functions, whenever the PHIGS structure id is known and the function must

communicate information about that structure to the server.

5. GetMStructMap(struct_id). This function takes as its parameter a PHIGS structure id. It finds that id in the hash table and then returns the list of (display, PEX structure id) pairs associated with this id. This is called when a new structure is opened, when an EXECUTE STRUCTURE structure element is output, when a structure is posted, or when a structure is deleted.

6. GetStructMapList(struct_id). This function takes a PHIGS structure id and returns the list of PEX structure ids that correspond to it. It is used by all functions that store structure elements.

7. GetInverseStructMap(pex_structure, display). This function takes as its parameters a PEX structure id and a display. It then searches the table looking for that pair in one of the (display, PEX structure id) lists. When it finds an entry in which both the display and the PEX structure id match, it returns the PHIGS structure id that is associated with that list. This is used by several functions that have a PEX structure id and need access to the corresponding PHIGS structure id.

8. DeleteStructMap(struct_id, dpy). This function searches the (display, PEX structure id) list that corresponds to the PHIGS structure id that is passed as a parameter. If the given display is found in the list, that item in the list is deleted. This is called when a structure is deleted in PHIGS.

9. DeleteMStructMap(struct_id). This function searches the table for the PHIGS structure id that is passed to it. When it finds it, the entire entry containing the PHIGS structure id and the (display, PEX structure id) list is deleted. This is called when a PHIGS structure network is deleted by a call to the PHIGS function *pdel_struct_net()*.

10. DeleteAllStructMaps(). This function deletes all of the entries from the hash table. It is called when the PHIGS function *pdel_all_struct()* is called.

11. GetNumStructsForDpy(display). This function determines the number of structures that are present for a given display. It is used to determine the number of structures to create on a new display when a window is opened on a display that is new to the system.

12. GetAllStructsForDpy(display). This function returns a list of all of the PEX structure ids that correspond to the given display. It is used for distributing structures to newly opened displays.

13. GetAllStructsForDpyList(display_list, num_displays). This function takes a list of displays and the length of the list. It then returns a list of elements, one element for each display in the given list. Each element in the list is a list of PEX structure ids for that display. It is used in distributing structures to displays when a new structure is opened.

CHAPTER 6.

Input

6.1. Input Overview

A PHIGS application obtains input values through the use of logical input devices. There are six types of logical input devices defined in PHIGS. These types, as well as the type of data they return to the application, are:

1. Locator: this device type returns a position in world coordinates and a view index.
2. Stroke: this device returns a sequence of points in world coordinates and a view index.
3. Valuator: this device type returns a real number.
4. Choice: this device type returns a positive integer that represents one selection from a list of possible choices.
5. Pick: this device type returns a PICK status, a pick path depth, and a pick path.
6. String: this device type returns a character string.

To use an input device, the application program specifies a logical input device type, a device number, and an open workstation. The API layer then creates an input device of that type, connects it with its corresponding physical device, and associates it with the specified workstation. User access to the input device is then done via the device number specified at device creation time.

Input devices can be in one of three *modes*. An input device is in exactly one mode at any given time. The input modes are *request*, *sample*, and *event*.

When an input device is in request mode and a REQUEST <input class> call is made, PHIGS waits until one of two events occurs. Either the user enters the input data (by typing a character, hitting a mouse button, etc.), or else a break is performed by the user. In the former case, the appropriate data for this input device type is returned to the user. In the case of a break being performed, the user is informed that no valid data is present.

When an input device is in sample mode and a SAMPLE <input class> call is made, PHIGS immediately returns the current value of the input device.

When an input device is in event mode and a call is made to retrieve an input event, the oldest input event is retrieved from the event queue and returned to the user. In addition, that event is removed from the queue.

6.2. Input Alternatives

There are two main ways in which input could be handled. One is to have an input event process that handles all of the input events that occur. The other way is to simply have the PHIGS process handle the events. The first method makes the handling of input more modularized, so that the input scheme could be enhanced or altered in the future. However, it introduces inefficiencies, since there must be interprocess communication in order for the input event process to inform PHIGS of modifications that need to be made due to the events it receives.

The second method is more tightly coupled with PHIGS, since it is running as part of the same process. However, it can be written in a modular way so that changes to it could be made in such a manner that would not require too much effort. This method also has the advantage of being very fast. Input events are reported to PHIGS via an interrupt scheme, and thus PHIGS can handle them in a very timely manner.

Since response time is of the essence when input is being performed, we chose the second alternative for our implementation.

6.3. Input Implementation

Whenever an X event occurs, an interrupt is generated. This causes the API level interrupt handler to be called. The interrupt handler then acts on events that are in the queue in a FIFO manner. The oldest event is taken from the queue. If it is an event that PHIGS is interested in, it performs the appropriate actions on that event, then goes to the queue and gets the next event. The events that are significant are **Structure Notify, Enter Window, Exposure, Leave Window, Focus Change, KeyPress, Button Press, Button Release,** and **Button Motion.** Of these event types, the first five report information about the window such as a change in window size, the mouse entering or leaving the window, etc. The remaining events have to do with actual input operations. These include pressing a key on the keyboard, pressing and releasing a mouse button, and moving the mouse while a button is pressed.

When an X event occurs, the following actions are preformed. First, the PEX server places the event on the event queue and makes a call to the API level interrupt handler. The interrupt handler then goes through the list of currently active displays. For each display, it looks at the list of X events. If there are events that have not been serviced, they are removed from the list one at a time and serviced at that time. This is done by finding the PHIGS workstation that corresponds to the X window in which the event occurred. Once that workstation has been determined, the X event can then be serviced. The event type is determined from the event itself, and the input devices for this workstation are checked to see if the event occurred within their echo areas. When an input device is found in which this event occurred, the input prompt is updated and any other relevant actions are performed (such as storing data received from the event, updating current input values, etc.).

At this point, this event has been serviced. The next event on the queue is then removed and serviced. This process is repeated until all events have been removed from the queue. Then, the event queue for the next display on the list is retrieved and its events are handled in a similar manner. This is done for each currently known display in the PHIGS list.

CHAPTER 7.

PHIGS Critique

PHIGS is gaining wide acceptance as a graphics standard. There are many capabilities in PHIGS that are very desirable. However, PHIGS also has its shortcomings. This chapter discusses some of the pros and cons of PHIGS.

7.1. Desirable Traits

7.1.1. 3D Output Primitives

The ability to model and view 3D objects is one of the PHIGS capabilities that is very useful. In fact, with today's demands for high quality graphics, it is essential. In order to produce an image that appears at all realistic, the system must allow an application to model objects in three dimensions. The extension from two dimensions to three dimensions in PHIGS was done quite cleanly. In addition to supporting the output primitives, other capabilities such as view orientations, view mappings (parallel or perspective), and depth cueing have been added as aids in viewing three dimensional objects. These traits will be present in all high level graphics systems in the future.

7.1.2. Hierarchical Modeling

The ability to organize objects into hierarchical entities provides the application with a tool that allows for easy reuse of existing objects. An object can be built up of several instances of another object simply by changing some of the attributes or the placement of that object. This ability saves the work of having to create an object each time it is to be instanced.

7.2. Undesirable Traits

7.2.1. Difficulty of Use

It takes a substantial amount of time for a person to learn how to use PHIGS and to be able to capitalize on the interesting features offered by PHIGS. A graphics system should be much more user friendly and easy to use. For example, to draw a picture of a chair, a user should not have to worry about the details of opening a workstation, opening a structure, setting the view orientation matrix (with the appropriate view reference point, view plane normal, and view up vector), setting the view mapping matrix (with the front plane, view plane, back plane, projection reference point, view window, and projection viewport), etc. He should be able to simply tell the system that he wishes to draw a chair.

PHIGS provides many capabilities that are useful, but are much too difficult for the average user to be able to use without a large amount of frustration and difficulty. In general, PHIGS programming is done at a very low level. There needs to be more abstraction away from minute details and toward higher level concepts.

7.2.2. Lack of Extensibility

When PHIGS was designed, it was designed as a graphics system that would provide a set of output primitives to be rendered in a scan conversion fashion. The ability to gracefully extend the system by an application was not a consideration. For example, an application must use only the output primitives supplied by the system. There is no provision for extending the set of output primitives available to an application. If a PHIGS application wishes to draw a circle, it must specify either a list of polylines or a fill area that will approximate a circle. The circle can be used by other objects, but it must always be the same circle. In order to draw a circle with a different radius, a new

polyline or fill area must be generated. Thus, the application is confined to the output primitives that the system supplies.

This is much too restrictive. A graphics system should allow an application the freedom to define its own output primitives, and then use those new primitives the same way it uses the system defined primitives. This would mean that applications could create primitives that would be pertinent to that application and reuse the new primitive as it saw fit, rather than having to redefine a new structure each time it wanted to instance the object.

The same principle is true for rendering in PHIGS. The PHIGS method of rendering allows for only one rendering technique to be used. A PHIGS application is forced to use whatever renderer is in the system, even though it may not be the appropriate renderer for that application. For example, some applications may wish to view their objects in a wireframe fashion, whereas other applications may wish to view their scene photorealistically. Furthermore, a single application may wish to use a fast, low quality renderer while the scene to be rendered is being set up and altered. Once the scene has been set up as the application desires, it may now wish to view the scene photorealistically. In PHIGS, there is no way of doing this. There is one mechanism for rendering a scene, and that is it. This is another feature of PHIGS that is too restrictive. A graphics system should provide a variety of rendering techniques, and should allow the application the freedom of extending the rendering system to include other techniques, as it sees fit.

7.2.3. Non-Geometric Modeling

Many graphics applications today wish to be able to store large amounts of information about objects, such as mass, density, surface type, material the object is made of, etc. In addition, it is sometimes desirable to be able to view these traits in some fashion. The only non-geometric attribute that is associated with output primitives in PHIGS is color information about the object. There is no

mechanism in PHIGS for storing other non-geometric attributes. This limits the number and type of applications that can easily use PHIGS.

CHAPTER 8.

Conclusions

As pointed out in the previous chapter, PHIGS possesses many characteristics that make it desirable for a wide range of applications. However, it also carries with it several deficiencies. This chapter presents methods of overcoming some of these deficiencies.

8.1. Making the System Extensible

One of the main problems with PHIGS is its lack of extensibility. A graphics system should allow an application the ability of creating its own modeling primitives and its own renderers. For example, an application may have volumetric data that it wishes to view. In PHIGS, this would be very difficult to do. The graphics system should allow an application the ability of adding this new modeling primitive and then integrate it smoothly into the existing system. Likewise, an application may wish to render its objects using a ray tracing scheme, so that the resulting image will be more realistic. PHIGS does not have this capability, and does not provide the application a mechanism for adding this to the system.

In addition to the ability of extending both the modeling and rendering portions of the graphics system, the system should be architected in such a way that these additions are as simple as possible, and require as little code change as possible. There are several methods of doing this.

One way of doing this would be to ensure that all renderers understood how to render all modeling primitives. When a new renderer is added to the system, it must be capable of understanding all modeling primitives that are then existent in the system. Whenever a new modeling primitive is to be added to the system, all renderers must be updated to understand this new

primitive. Since the number of renderers will probably be very small (3 or 4), and the number of modeling primitives will probably be much greater (15 or 20), then it seems to make sense to update the renderers, since there are fewer of them. However, what we really would like is a system where new modeling primitives can be added without altering the existing renderers, and new renderers can be added without altering the existing modeling primitives.

This can be done by having a common data type that all renderers can understand. When an object is to be rendered, it must convert itself into this common type, and then send the resulting data to the renderer. Adding new modeling primitives is simple, since no changes to existing renderers need be made. The new object is simply defined, as well as a conversion routine to convert the object into the common type. Adding new renderers is also simple, since no changes need to be made to existing modeling primitives. The new renderer only needs to be able to accept input data that is of the common type, and then render that data.

This scheme has one major drawback, however. Converting all primitives into one common type and having all renderers render that type will be very inefficient. Some rendering techniques are ideally suited for a particular data type or types, and can render them very quickly. Converting objects into a common type will preclude renderers from using many of the efficient features that are built into them.

To overcome this problem, the modeling objects and renderers must have some form of communication between them. The renderers can then inform the modeling objects of the type of data that they would prefer to have sent to them, and the modeling objects can then possibly send that more efficient data type. This allows renderers to use their efficiencies, but still provide an extensible system. There must still be a common primitive that all renderers can render and that all modeling objects can transform themselves into, but now the system is more general and can capitalize on the efficiencies of the renderers.

8.2. Allowing for Composite Rendering

Another capability that a graphics system should have is the ability to render different portions of a scene in different ways. For example, we may wish to view a car on a country road, where we wish the car to be rendered photorealistically, the country side to be rendered as shaded images, but not necessarily photorealistically, and the fences around the fields to be rendered in a wireframe fashion. PHIGS does not have this capability, nor does the author know of any system that easily allows this to be done. This functionality would allow an application the ability of selecting the objects that it would like the system to spend the most time on, and the objects that can be rendered in a faster, less realistic fashion.

One possible method for implementing this capability is to have various levels at which a renderer can render. When an object is to be rendered, it tells the renderer at which level to render. The renderer then renders it accordingly.

As modeling objects are built up in the modeling portion of the system, and as new objects are added to the system, each component of a modeling object has associated with it a rendering level. When a composite object is to be rendered, it sends each of its components, along with their respective levels, to the renderer. The renderer then renders each part at its associated level.

By using this technique, a scene can be built up as an object that is comprised of many parts. When the application wishes to view the scene, it simply renders this object. This in turn causes each of the component parts to be sent to the renderer, along with their rendering levels, to be rendered.

8.3. Object-Oriented Programming

The ideas discussed in this chapter suggest that the graphics system proposed here be implemented in an object-oriented environment. Using this style of programming would give us the

inheritance and code reuse that is needed in defining new modeling objects and renderers. It would also give a clean communication interface between modeling objects and renderers, which is needed not only for the rendering phase itself, but also so that renderers can inform modeling objects about their capabilities.

Object-oriented programming would also provide a suitable environment for the implementation of composite rendering. Objects could be made up of instances of other objects, and thus benefit from work already done in the system, and renderers could possibly use portions from previously created renderers so that a renderer to be added to the system need not be built from scratch each time.

References

[1] American National Standards Institute, ‘‘Computer Graphics - Graphical Kernel System (GKS) Functional Description,’’ X3.124-1985.

[2] American National Standards Institute, ‘‘Computer Graphics - Programmer’s Hierarchical Interactive Graphics System (PHIGS) C Binding,’’ Draft Proposed Standard X3.1444-1988, June 1988.

[3] American National Standards Institute, ‘‘Computer Graphics - Programmer’s Hierarchical Interactive Graphics System (PHIGS) Functional Description, Archive File Format, Clear-Text Encoding of Archive File,’’ X3H3/89-54, September 26, 1988.

[4] American National Standards Institute, ‘‘Computer Graphics - Programmer’s Hierarchical Interactive Graphics System (PHIGS) Part 4 - Plus Lumîere und Surfaces (PHIGS PLUS),’’ Draft Proposed Standard, X3H31-89-05, July 25, 1989.

[5] Clifford, William H. Jr., John I. McConnell, and Jeffrey S. Saltz, ‘‘The Development of PEX, a 3D Graphics Extension to X11,’’ *EUROGRAPHICS ’88, Proceedings of the European Computer Graphics Conference and Exhibition*, Nice, France, September 12-16, 1988, pp. 21-30.

[6] Egbert, Parris K., ‘‘UIPEX - PHIGS PLUS C Language Binding,’’ University of Illinois at Urbana-Champaign, Department of Computer Science, Graphics Workstation Research Group Technical Report No. 4, March 1990.

[7] Egbert, Parris K., ‘‘UIPEX - Application Programmer Interface Design Documentation,’’ University of Illinois at Urbana-Champaign, Department of Computer Science, Graphics

Workstation Research Group Technical Report No. 6, March 1990.

[8] Newman, T. ‘‘PEX Protocol Encoding, Version 3.30,’’ Digital Equipment Corporation, Workstation Systems Engineering, July, 1989.

[9] Rost, R.J., ‘‘Adding a Dimension to X,’’ *UNIX Review*, vol. 6, no. 10, October 1988, pp. 51-59.

[10] Rost, R.J., ‘‘PEX Introduction and Overview, Version 3.20,’’ Digital Equipment Corporation, Workstation Systems Engineering, October, 1988.

[11] Rost, R.J., J.D. Friedberg, and P.L. Nishimoto, ‘‘PEX: A Network-Transparent 3D Graphics System,’’ *IEEE Computer Graphics & Applications*, July 1989, pp. 14-26.

[12] Rost, R.J., et. al., ‘‘PEX Protocol Specification, Version 3.30,’’ PEX Architecture Team, June, 1989.

[13] Scheifler, R. W. and J. Gettys, ‘‘The X Window System,’’ *ACM Transactions on Graphics*, vol. 5, no. 2, April 1986, pp. 79-109.

[14] Sung, Kelvin H. C., ‘‘UIPEX: A 3D Graphics Extension to the X Window System,’’ Master’s Thesis, Report UIUCDCS-R-90-1584, Department of Computer Science, University of Illinois, 1990.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor W. J. Kubitz for his suggestions and encouragement. His insight and directions have been most beneficial throughout the duaration of this project.

I would also like to express my appreciation to Professor R. H. Campbell, the other overseer of the Graphics Workstation Research Group, for his ideas and input. In addition, I would like to thank the other members of the group: Greg Rogers who proved to be an invaluable resource throughout the project, Kelvin Sung who impelemented the DIPEX layer, Bill Brown who implemented the ME layer, and David Griffith who wrote the DDPEX portion of the system. Thanks also goes to the International Business Machines Corporation, which partially funded this project.

Last, but not least, I would like to thank my wife, Lori, and the rest of my family for their much needed patience and support.

TABLE OF CONTENTS

CHAPTER

1. Introduction 1

2. System Overview 3
   2.1. PHIGS 3
   2.2. PEX 6

3. The Application Programmer Interface Level 10
   3.1. Manipulating PHIGS Data 11
   3.2. Retrieving Data from the Server 11
   3.3. Error Checking 12
   3.4. Error Handling 13
   3.5. Maintaining State Information 14
   3.6. Input 15

4. Multiple Server Architecture 17

5. Structure Mapping 21
   5.1. Opening a New Structure 23
   5.2. Opening a New Display 24
   5.3. Reasons Behind This Design 24
   5.4. Operations on the Structure Mapping Table 25

6. Input 28
   6.1. Input Overview 28
   6.2. Input Alternatives 29
   6.3. Input Implementation 30

7. PHIGS Critique 32
   7.1. Desirable Traits 32
   7.2. Undesirable Traits 33

8. Conclusions 36
   8.1. Making the System Extensible 36
   8.2. Allowing for Composite Rendering 38
   8.3. Object-Oriented Programming 38

References 40