

Based on basic graphic principles compare PEXlib 5.2 and OpenGL 1.0
by Chongyu Zhou

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Chongyu Zhou (1996)

Abstract:
Computer graphics bring to life a vast range of human imaginings. Graphics and visualization become unprecedented power to explore our world. Over the past decade, the computer's ability to create images has led to such innovations as mouse-driven graphical user interfaces, desktop publishing and multimedia communications. The entertainment industry now relies on visual computing for vivid new characters and landscapes. Through computer imaging inside the human body can be seen without performing surgery. For many scientists, doctors, engineers and other professionals, the computer's power to visualize both real world data and abstract relationships has become an indispensable tool.

Although there are a lot of 3D graphics packages or 3D application program interfaces (APIs), the PEXlib and OpenGL play very important roles in the 3D API market and workstation market. To understand them is also important for users to create their scenes by using a reasonable 3D API.

Since the PEXlib 5.2 and OpenGL 1.0 are two complex 3D API packages, it is difficult to explore and compare them completely from user's side. In this research, the comparison of PEXlib 5.2 and OpenGL 1.0 based on the basic principles of graphics is done. In general, the comparison between them follows the basic concepts for 3D graphics, such as coloring, lighting, shading, viewing and their architectures. The results of this comparison are not as simple as answering yes or no questions. Hopefully, in some areas this thesis will help users to choose the right API for their applications by using computer efficiently because the right decision can get right results quickly most of the time.

BASED ON BASIC GRAPHIC PRINCIPLES

COMPARE PEXLIB 5.2 AND OPENGL 1.0

by

Chongyu Zhou

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY-BOZEMAN
Bozeman, Montana

July 1996

N378
Z6156

APPROVAL

of a thesis submitted by

Chongyu Zhou

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

7/31/96
Date

J Dunbigh Stanley
Chairperson, Graduate Committee

Approved for the Major Department

7/31/96
Date

J Dunbigh Stanley
Head, Major Department

Approved for the College of Graduate Studies

8/1/96
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University-Bozeman, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Zhou Chongzu

Date 7/30/96

ACKNOWLEDGMENTS

I wish to express my deepest gratitude to Dr. J. Denbigh Starkey for his supervision, advice and support through the course of this research. I would also like to thank Professor Ray Babcock and Dr. Bob Cimikowski for their kind guidance and serving on my committee.

Thanks are extended to Adonis El Fakih who works for the Engineering Graphical Solutions Company for his assistance and experiences with both PEXlib and OpenGL.

Finally, I would also like to thank Jim Costello who is a sales representative in Silicon Graphics for his supporting some important information of this study.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT xi

1. INTRODUCTION 1

Graphics Standards 1
Motivation and Objectives 3

2. OVERVIEW 5

Overview of PEXlib 5.2 5
PEX and PEXlib 5
History of PEXlib 6
Version and Characteristics for PEXlib 8
Overview of OpenGL 1.0 10
History of OpenGL 10
Characteristics of OPenGL 1.0 10
GLX 12
Applications and Version for OpenGL 14

3. ARCHITECTURE 15

The Architecture of PEXlib 5.2 15
Immediate and Structure Modes 16
PEX Resources 17
Summary of the Architecture for PEXlib 5.2 25
The Architecture of OpenGL 1.0 25
Architecture's Structure 26
Framebuffer 29
Immediate Mode in OpenGL 1.0 31
Display List Mode 32
Summary of the Architecture for OpenGL 1.0 35
Analysis 35

4. COLORING 37

Color Models 37
RGB Model 38
Advantages For Using Color 39

TABLE OF CONTENTS—Continued

Coloring in PEXlib 5.2 40
Color Models in PEXlib 5.2 41
Intrinsic Color 41
Color Pipeline 42
Indexed Color 44
Color Table 46
Summary of the Coloring System in PEXlib 5.2 47
Coloring in OpenGL 1.0 47
Color Modes 48
RGBA Mode 49
Color-Index Mode 51
Analysis 52

5. LIGHTING 53

Basic Principles of Lighting 54
Types of Light 54
Lighting in PEXlib 5.2 58
Types of Light Sources in PEXlib 5.2 58
Reflection 64
Summary of the Lighting System in PEXlib 5.2 68
Lighting in OpenGL 1.0 69
Types of Light Sources in OpenGL 1.0 69
Lighting Properties 71
Lighting in Color-index Mode 73
Summary of the Lighting System in OpenGL 1.0 74
Analysis 74

6. SHADING 76

Basic Principles of Shading 76
Constant Shading 77
Gouraud Shading 78
Phong Shading 81
Summary of Gouraud and Phong Shading 82
Shading in PEXlib 5.2 83
Summary of the Shading System in PEXlib 5.2 85
Shading in OpenGL 1.0 85
Summary of the Shading System in OpenGL 1.0 87
Analysis 87

7. VIEWING 88

Basic Principles of Viewing 88
Virtual Camera 89
Projection 90
Coordinate System 93
Steps for Defining a View 95

TABLE OF CONTENTS—Continued

Viewing in PEXlib 5.2..... 96
Summary of the Viewing System in PEXlib 5.2..... 99
Viewing in OpenGL 1.0.....100
Analysis.....102

8. MISCELLANEOUS FEATURES AND PORTABILITY.....103

Miscellaneous Features for PEXlib 5.2
and OpenGL 1.0.....103
Portability.....105

9. CONCLUSIONS AND RECOMMENDATIONS.....108
Conclusions.....108
Recommendations.....110

REFERENCES CITED.....111

LIST OF TABLES

Table Page

1. PEXlib 5.2 Functional Categories..... 9
2. PEX 5.2 Light Types..... 59
3. Three Types of Reflection in PEX 5.2..... 65
4. The Relationships Between the Light Types and the Reflection Types in PEXlib 5.2..... 67
5. Correspondence of Lighting Parameter Symbols to Names. AMBIENT_AND_DIFFUSE Is Used to Set a_{cm} and d_{cm} to the Same Value..... 71
6. Lighting Properties Defined by OpenGL 1.0..... 72
7. Registered Shading Methods in PEXlib 5.2..... 84
8. How OpenGL 1.0 Selects a Color for the ith Flat-Shading Polygon..... 86
9. Steps Used to Define a View in a 3D System..... 96
10. The Miscellaneous Features for PEXlib 5.2 and OpenGL 1.0.....104

LIST OF FIGURES

Figure Page

1. User Interface Library..... 6
2. OpenGL can be implemented as an extension to several operating and window system standard environments..... 11
3. GLX client, X server and Framebuffer..... 14
4. The primitives and attributes travels in PEX..... 16
5. The Renderer..... 18
6. Conceptual Diagram of a PEX Lookup Table..... 19
7. A Structure..... 20
8. Structure Network..... 21
9. Geometry transformation stages of the rendering pipeline..... 22
10. Architecture of OpenGL 1.0..... 27
11. Per-Fragment Operations..... 28
12. The RGB Cube..... 38
13. Direct Color and Indexed Color..... 40
14. Stage of Color Pipeline..... 43
15. Processing for Directcolor, Pesudocolor and Grayscale visuals..... 43
16. PEX resolves an indexed color by selecting the color table entry for the index..... 45
17. A Color Table..... 46
18. Processing of Colors in OpenGL 1.0..... 49
19. RGB Values from the Bitplanes..... 50

LIST OF FIGURES—Continued

Figure Page

20. Five Light Types..... 55
21. The PEX 5.2 Light Sources..... 59
22. The Parameters of A Spotlight..... 62
23. The Normals of a Truly Curved Surface..... 78
24. Normalized polygon surface normals may be averaged to obtain vertex normals..... 79
25. Intensity Interpolation Along Polygon Edges and Scan Lines..... 80
26. A Polygon that `Contains' A Highlight..... 81
27. In the Phong method vector interpolation replaces intensity interpolation..... 82
28. Gouraud and Phong Shading..... 83
29. Conceptual Model of The 3D Viewing Process..... 88
30. A Camera with (U,V,N) System..... 89
31. Principle of Parallel Projection..... 91
32. Principle of Perspective Projection..... 91
33. The view plane is defined by VPN and VRP..... 92
34. The View Plane..... 93
35. Coordinate System and How they relate to one another..... 94
36. The View Reference Coordinate System..... 94
37. Right-handed Coordinate System..... 95
38. View Table..... 97

ABSTRACT

Computer graphics bring to life a vast range of human imaginings. Graphics and visualization become unprecedented power to explore our world. Over the past decade, the computer's ability to create images has led to such innovations as mouse-driven graphical user interfaces, desktop publishing and multimedia communications. The entertainment industry now relies on visual computing for vivid new characters and landscapes. Through computer imaging inside the human body can be seen without performing surgery. For many scientists, doctors, engineers and other professionals, the computer's power to visualize both real world data and abstract relationships has become an indispensable tool.

Although there are a lot of 3D graphics packages or 3D application program interfaces (APIs), the PEXlib and OpenGL play very important roles in the 3D API market and workstation market. To understand them is also important for users to create their scenes by using a reasonable 3D API.

Since the PEXlib 5.2 and OpenGL 1.0 are two complex 3D API packages, it is difficult to explore and compare them completely from user's side. In this research, the comparison of PEXlib 5.2 and OpenGL 1.0 based on the basic principles of graphics is done. In general, the comparison between them follows the basic concepts for 3D graphics, such as coloring, lighting, shading, viewing and their architectures. The results of this comparison are not as simple as answering yes or no questions. Hopefully, in some areas this thesis will help users to choose the right API for their applications by using computer efficiently because the right decision can get right results quickly most of the time.

CHAPTER 1

INTRODUCTION

Computer graphics is a branch of computer science. In particular, computer graphics is mainly deals with 3D graphics and interactive 3D graphics. Today it has been used in many different areas of industry, business, government, education, and entertainment, etc. The list of applications is enormous, and is growing rapidly as computers with graphics capabilities become commodity products[1].

Graphics Standards

As the interest in computer graphics has grown, so has the desire to be able to write an application so that it runs on a variety platforms with a wide range of graphical capabilities. A graphics standard eases this task by eliminating the need to write a distinct graphics driver for each platform on which the application is to run. Emerging standards for 3D graphics hardware and software are leaving the relatively obscure realms of scientific visualization, engineering and computer animation and appearing in ordinary operating systems and applications.

For 2D graphics, several standards have succeeded. For example, the PostScript page description language has become widely accepted, making it relatively easy to electronically exchange, and, to a limited degree, manipulate static documents containing both text and 2D graphics. Also, the X Window System, which was developed by Paul Asente and Brian Reid in 1984, has become standard for UNIX workstations. A programmer uses X to obtain a window on a graphics display into which either text or 2D graphics may be drawn; X also provides a standard means for obtaining user input from such devices as keyboards and mice. The adoption of X by most workstation manufactures means that a single program can produce 2D graphics or obtain user input on a variety of workstations by simply recompiling the program. This integration even works across a network: the program may run on one workstation but display on and obtain user input from another, even if the workstations on either end of the network are made by different companies.

For 3D graphics, several standards have been proposed, but none has gained wide acceptance[2]. 3D graphics standards include APIs (Application Program Interfaces) that define standard sets of graphics functions for programmers to use, and specialized hardware to accelerate those functions, as well as file formats to help pass data from one program to another. There are several standards that have been proposed but none has gained wide acceptance. One

relatively well-known system is PHIGS (Programmer's Hierarchical Interactive Graphics System). Based on GKS (Graphics Kernel System), PHIGS is an ANSI (American National Standards Institute) standard [3]. Workstation 3D graphics are dominated today by two competing standards: OpenGL from Silicon Graphics Inc., and PEXlib, from a consortium based at MIT (Massachusetts Institute of Technology) that has developed PEX (PHIGS, PHIGS+, and Extensions to X), a set of 3D extensions to the X Window System.

PEXlib and OpenGL are both receiving broad support as industry standards. The PEX standard is promoted by HP, SUN, IBM and all members of COSE (Common Operating System Environment) which has over 70 members. On the other hand, the OpenGL standard is supported by 35 vendors, such as AT&T, DEC, IBM and HP[4]. Also, the OpenGL standard is maintained by an independent group called the ARB (Architectural Review Board), which meets quarterly and includes representatives from DEC, IBM, Intel, Microsoft, Evans and Sutherland, Intergraph and Silicon Graphics.

Motivation and Objectives

The purpose of this thesis is to compare PEXlib 5.2 and OpenGL 1.0 which are considered as major 3D graphics APIs in industry and education. Each of them combines with its own

architecture, color system, lighting system, shading system, viewing system and some special features that are different from each other. The main focus of comparison for those two systems is based on the main functions that they contain. The goal of the comparison is to explore more details for both systems that give users some ideas about how to choose their convenient graphics API to fit their applications which may change in thousands of ways. Also, this comparison is an objective description and based on the basic graphic principles and the view from user's side. Since PEXlib 5.2 and OpenGL 1.0 are two complex packages, it is hard to compare them completely. Hopefully the goal can be reached after this thesis is presented.

CHAPTER 2

OVERVIEW

Overview of PEXlib 5.2

PEX and PEXlib

PEXlib is the richest 3D graphics programmer's interface which lets users write their programs using the PEX protocol and PEX is the formal graphics protocol. PEXlib is designed as an extension to Xlib and uses the same transport and error mechanisms as Xlib and provides vendor independence, even over a number of networked system from different vendors, using the client-server model used by all X releases as shown in Figure 1. This particular definition of PEXlib is intended for using PEXlib only in the X Window System environment. In addition, some companies are trying to bring PEX to Microsoft Windows.

When discussing PEX, it is important not to confuse the protocol with the API. The API is the conceptual model of 3D graphics that the application developer sees. The protocol

is generated by the API and is interpreted by the server to perform graphics requests on behalf of the client program.

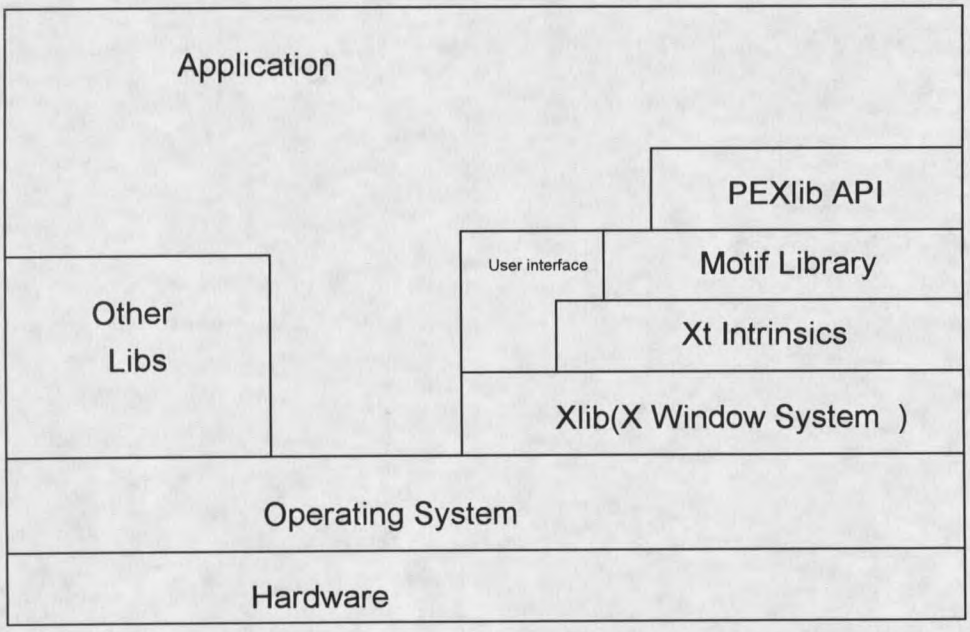


Figure 1. User Interfaces Library

History of PEXlib

A demonstration implementation of PEX was produced and shown at the SIGGRAPH'88 vendor exhibition in Aug. 1988[5]. PEX originally stood for PHIGS Extensions to X11, network protocol which implements the X window system because it was once intended to be used for implementing PHIGS in the X windows environment due to the popularity of PHIGS at that time. As time passed, PHIGS support becomes less interesting, and the goal of supporting PHIGS became

secondary to provide support for 3D application programs in general. Today's PEX contains features well beyond those which need to support PHIGS.

PEXlib is the first computer graphics standard to be truly portable [6]. The promise of portability offered with GKS and PHIGS was never kept, due to each of the vendors developing libraries independently. Often, PHIGS products were based on different drafts of the standard offering slightly different features and interfaces. The PEXlib source is distributed by the X consortium at MIT virtually free of charge. Given that all implementations of the PEXlib standard start with the same code, they have identical interface to the standard. The implementations will vary in extensions, but even that is often shared among vendors.

In March 1993, most of the major graphics workstation companies outlined the Common Open Software Environment (COSE). This is a collection of user interfaces and programming interfaces, known as the Common Desktop Environment (CDE), to make the UNIX systems look and feel more consistent. PEXlib was chosen as the 3D graphics API, an indication that PEXlib is the best choice for an API for the X window system and UNIX environments[7].

Version and Characteristics for PEXlib

The latest version of PEXlib was named PEXlib 5.2 and next will be PEXlib 6.0. The PEXlib 5.2 library contains over 250 powerful functions for 3D graphics developers. These provide a large amount of functions for users to design their graphic images. The functional categories of PEXlib 5.2 are listed in Table 1. The PEXlib 5.2 specification currently defines function bindings and include files for the C language only [8]. The specification is intended to work for both ANSI C and Kernighan and Ritchie (K & R) C compilers. Also, if the major-version of the PEX extension's protocol is different than 5, then PEXlib 5.2 won't work with this PEX extension. A PEXlib 5.2 client works with a PEX extension that has a minor-version level of 2 or greater. If the minor-version level is 0 or 1, then only a subset of the functions described in PEXlib 5.2 specifications is available. On the other hand, there are some limitations for PEXlib 5.2, it doesn't do ray tracing or radiosity. Neither does it compute shadows or follow light as it bounces from one object to another.

Table 1. PEXlib 5.2 Functional Categories

Categories	Contents
1	PEXlib Access and Information Functions
2	OC Context Functions
3	Primitives and Attribute Binding
4	Output Primitive Functions
5	Structure Control and Miscellaneous Functions
6	Lookup Tables
7	Pipeline Contexts
8	Renderers
9	Renderer Picking
10	Structures
11	Name Sets
12	Texture Mapping
13	Accumulation Buffers
14	Spatial Searches
15	PEX Fonts
16	PEX Escapes
17	PHIGS Workstation Picking
18	Output Command Encode and Decode
19	Sending Encoded Output Commands

Overview of OpenGL 1.0

History of OpenGL

OpenGL began as IRIS GL, which was a proprietary graphics programming interface for Silicon Graphics workstations and IRIS GL is heavily used in the animation industry dominated by Silicon Graphics hardware[9]. Years of graphics programmers working with some of the most advanced graphics hardware and software refined the character of IRIS GL. As IRIS GL became more popular and customers wanted to port their GL (Graphics Language) programs to other platforms, Silicon Graphics opened the GL standard to other parties, giving rise to OpenGL. OpenGL was announced in Sep., 1991.

Now, OpenGL is a new and emerging graphics standard. It became a registered trademark in January 6, 1995. Until December 1994, SGI has issued more than 2,000 commercial OpenGL licenses and another 10,000 university licenses, according to Mason Woo, product marketing manager for OpenGL at SGI, Mountain View.

Characteristics of OpenGL 1.0

OpenGL 1.0 is a software interface that allows graphics programmers to create high-quality 3D graphic images

complete with shading, lighting, and other effects [10]. OpenGL 1.0 is an immediate mode system, that is commands are executed essentially immediately. OpenGL 1.0 accepts various data types, freeing the programmer from needless conversions. It is procedural in that the programmer issues specific commands to determine what is actually drawn. And it supports non-editable display lists for better network extensibility, resulting in improved performance across server/client applications, as well as improved direct rendering. OpenGL 1.0 doesn't have functions related to windowing, so it is not tied to other standards like X Windows or Microsoft Windows. Neither does it depend on the operation system, network or hardware (see Figure 2). Like PEX, OpenGL integrates 3D drawing into X, but it can also be integrated into other window systems (e.g., Windows/NT) or can also be used without a window system [11].

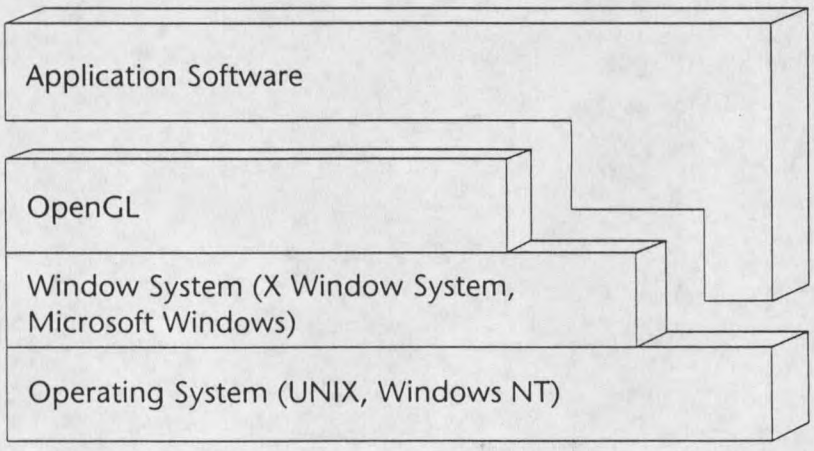


Figure 2.OpenGL can be implement as an extension to several operating and window system standard environments.

OpenGL 1.0 provides a wide range of graphics features. The 250 routines of OpenGL 1.0 let software developers access approximately 120 distinct commands to draw various primitives such as points, lines, and polygons. OpenGL 1.0 also includes support for shading, texture mapping, anti-aliasing, lighting, and animation which is the same as PEXlib 5.2 provides, in addition to support for atmospheric effects such as fogging and simulation of depth-of-field. Also OpenGL 1.0 provides language binding features so that it can be called from C, C++, FORTRAN, Ada and Java. OpenGL 1.0 is available for environments including Windows NT 3.5, Win 32s, OS/2, Sun Solaris, Apple MacOS for Power Macintosh, Digital Equipment OSF/1 AXP and many others[12]. In other words, OpenGL 1.0 not only can be run on workstations, but also can be run on the PC environments.

GLX

The specification which describes how OpenGL 1.0 integrates with the X Window System is known as GLX. It is an extension to the core X protocol for communicating OpenGL commands to the X server. It also supports window system specific operations such as creating rendering contexts, binding those contexts to windows and other window system specific operations. At this point, OpenGL implementations

allow mixed X and OpenGL rendering in a single window which is a capability PEXlib 5.2 does not provide[13].

GLX does not demand OpenGL 1.0 commands be executed by the server. The GLX specification explicitly allows OpenGL to render directly to the hardware if supported by the implementation. This is possible when the program is running on the same machine as the graphics hardware. This potentially allows extremely high performance rendering because OpenGL 1.0 commands do not need to be sent through the X server to get to the graphics hardware, which is shown in Figure 3. While GLX works alongside the X Window System, it also has a much more sophisticated concept of what the graphics hardware can do. X treats the fundamental graphics canvas as an array of 2D pixels, but GLX expands this by including capabilities of the OpenGL 1.0 framebuffer which is also shown in Figure 3. So models like stereoscopic viewing and double buffering that are useful for animation can be supported. This feature requires the graphics hardware to contain a framebuffer, and it is a reasonable requirement since nearly all interactive graphics applications run on systems with framebuffers [14].

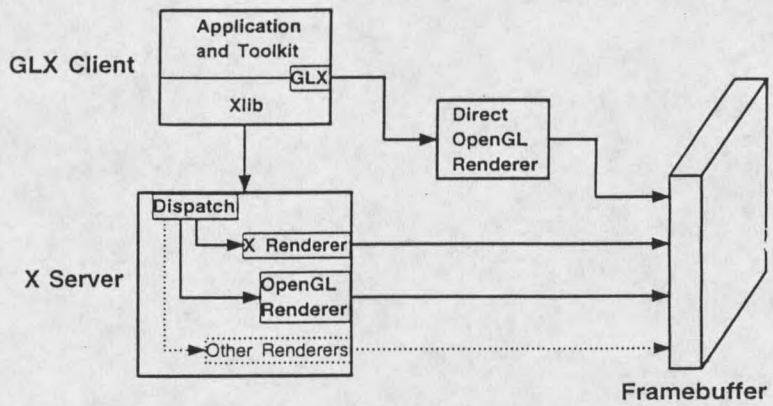


Figure 3. GLX client, X server and Framebuffer

Applications and Version for OpenGL

OpenGL has a strong presence in 3D graphics applications in industry. For example, OpenGL technology was used to provide special effects in films such as Jurassic Park and Terminator 2, as well as various television series such as Babylon 5[15]. Another area is the World-Wide Web, which the VRML (Virtual Reality Markup Language) standard based in part on OpenGL makes it possible to include 3D scenes and objects in Web pages. Although, OpenGL is powerful in many areas such as film animation and scientific visualization, it isn't perfect-it doesn't easily do shadows, lacks text functions, and has speed problem in PC. The latest version of OpenGL is OpenGL 1.1 which was specified in February, 1996. The newest version of GLX will be GLX 1.3 which will be released in the summer of 1996.

CHAPTER 3

ARCHITECTURE

The Architecture of PEXlib 5.2

Because PEXlib 5.2 is a high-level graphics library, it allows users to describe a graphic image in terms of familiar objects and attributes, without having to deal with the details of producing that graphic image in a window. However PEX 5.2 takes the output commands from applications, performs a number of operations on them, and ultimately computes the pixel values. The primitives and attributes travel to PEX sequentially, as given in Figure 4 [16]. Pictures are drawn in PEX a frame at a time. A frame is simply a convenient term for a sequence of output commands. A frame might describe a static image, as does a single frame of film, or it might be an interactive frame, where the users of the programs indicate piece by piece what to add to the picture.

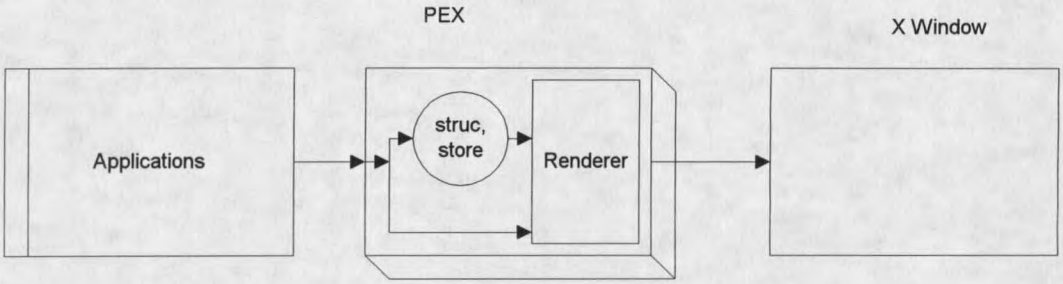


Figure 4. The primitives and attributes travels in PEX

Immediate and Structure Modes

The PEX 5.2 specification provides two ways to draw graphic images. One is immediate mode where primitives are drawn as they are sent to the server. It means passing all the primitives in the scene to PEXlib 5.2 one by one and having PEX 5.2 display them immediately. Immediate mode lets you directly interact with geometric representation of an object via the display on the screen. The advantages and disadvantages of immediate mode will be discussed later. This mode is best for applications that must continuously re-specify the complete graphic image, such as animated simulation programs that display different geometry with each frame.

The second mode supported by PEX 5.2 is structure mode. In this mode, graphics primitives are loaded into the structure store, and then a render structure request is sent to the server which renders all the data in store. It means

users don't need to send the primitives for each redisplay, but merely tell PEX to redisplay the stored database. Also, users can selectively edit the database between redisplay, changing only those parts of the picture that should be different in each scene. The primitives and their attributes are stored in the database. Structure mode is best for programs such as CAD (Computer Aided Design) programs that create a graphics model and edit portions of it frequently or view it in different ways. It is also possible to mix both immediate mode and structure mode. Programmers can store the static or infrequently changed parts of their model in structures, and draw the other parts using immediate mode.

PEX Resources

Like X itself, the PEX architecture is object-oriented, creating an environment that is flexible as well as extensible. Clients can create, free and manipulate objects called resources. Partitioning the desired functionality into resource type was a difficult task [17]. The major components which PEX 5.2 provides are output commands, drawables and their ancillary buffers, and the PEX resources: renderers, pipeline contexts, lookup tables, name sets, structures, pick measures, PHIGS workstations, fonts and search contexts.

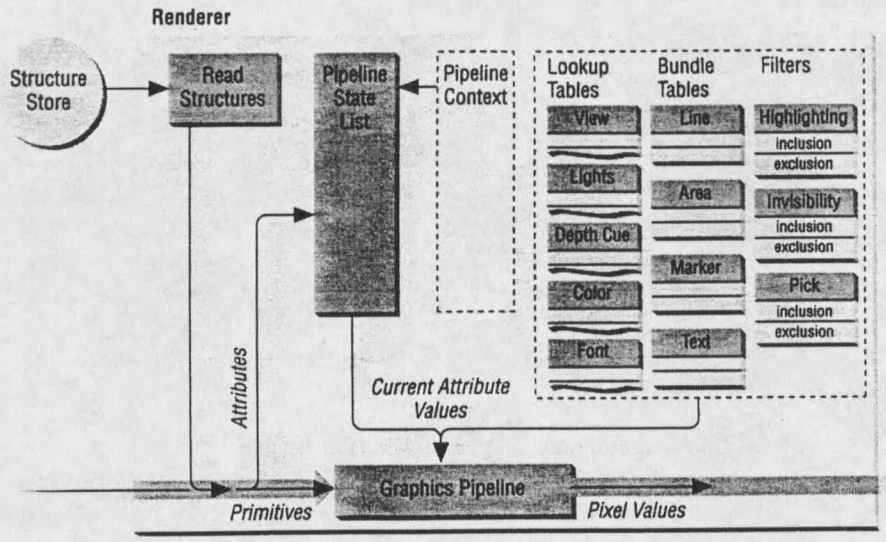


Figure 5. The Renderer

The renderer is the central PEX component, as shown in Figure 5. It transforms output commands representing a 3D model to a 2D image drawn in a drawable. Most of the other PEX components are bound to a renderer to accomplish this transformation. Each renderer has attributes which affect its rendering. User must specify a renderer in order to draw PEX graphics. In fact, the renderer is bound to the drawable at the beginning of a sequence of rendering requests and is not unbound until the sequence has completed. The renderer draws 3D geometrical objects into the X drawable bound to it. The type (e.g., window, pixmap) and visual identifier of those drawables are supported by PEX for a given screen, depth and visual. The EVEX (Extended Visual Extension to X) is recommended for inquiring the drawable's frame buffer capabilities, including which ancillary buffers are

supported. PEX uses the accumulation, alpha and Z-buffer for its ancillary buffers. Also it is recommended the MBEX (Multi-buffer Extension to X) be used for double-buffering. The MBEX image buffers are alternately bound to a renderer to achieve this effect.

Pipeline contexts are PEX resources containing an instance of the set of attributes which describe a rendering pipeline state. In other words, pipeline contexts are used to provide the initial state for the PEX rendering pipeline. Every attribute that affects the behavior of the rendering pipeline is defined as an attribute of the pipeline context. Although many pipeline context resources may exist, at any particular time at most one is associated with a particular renderer. Also, a pipeline context is associated with a renderer by setting the renderer's pipeline context attribute to the resource ID of the pipeline context.

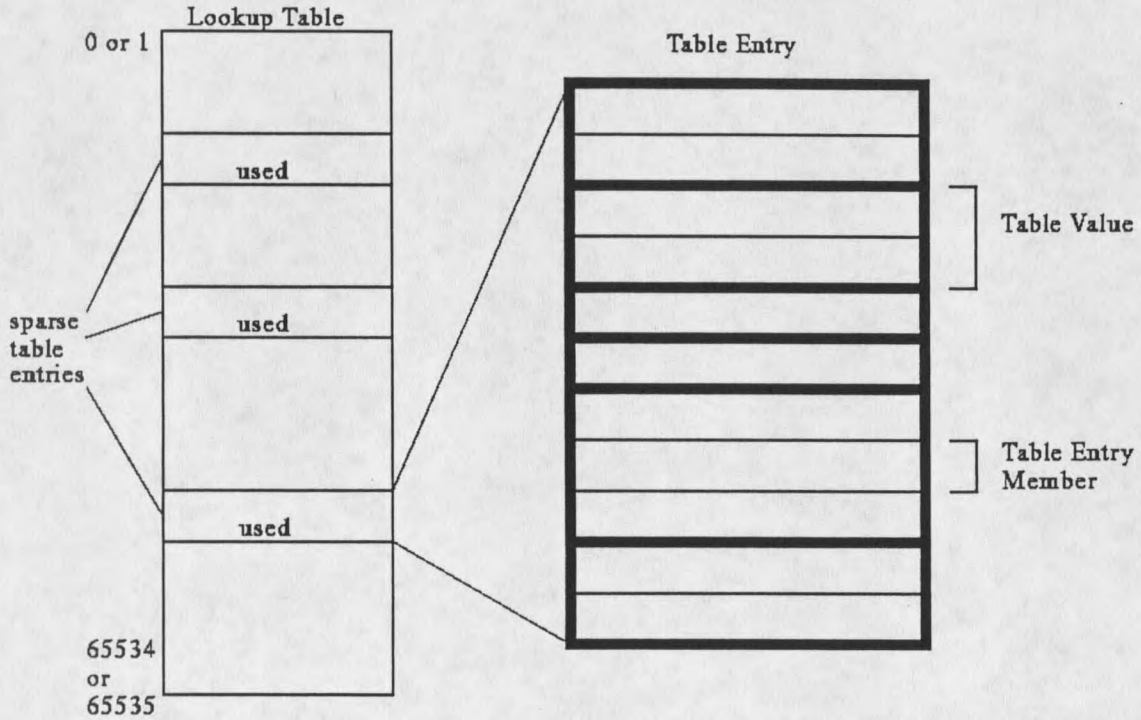


Figure 6. Concept Diagram of a PEX Lookup Table

Lookup tables which are shown in Figure 6 are used to maintain lists of attributes, such as those used for viewing, depth-cueing, illumination computations and defining the appearance of output primitives. In short, a lookup table holds collections of related values. When processing output commands, the renderer accesses the data that users specify in the lookup tables which are user defined. It may be easier and faster for users to change values in a lookup table, where users group values, rather than editing every single instance of data which the user wants to change throughout a large structure network. Lookup tables are also used in programs to store information, such as light source description, depth-cueing parameters and view definitions.

Name set resources contain arbitrary-length lists of identifiers that can be used to provide conditional control over such operations as highlighting, visibility, structure searching and detectability for picking purposes.

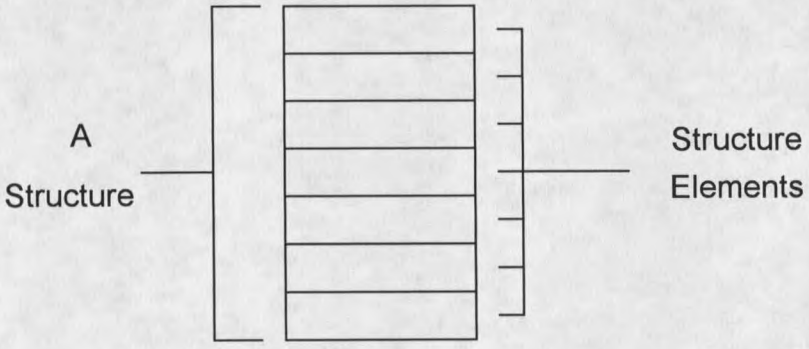


Figure 7. A Structure

A structure is a PEX resource which stores output commands for later execution. A simple structure is shown in Figure 7. The PEX client program places output commands in a structure where they are referred to as structure elements. Since PEX structures can call other structures, such a hierarchy of structures is called a structure network which is shown in Figure 8. Structure networks can be represented as acyclic graphs, therefore structures should not reference structures recursively, either directly or indirectly. Once a PEX structure is created, it may be edited by inserting or deleting new structure elements.

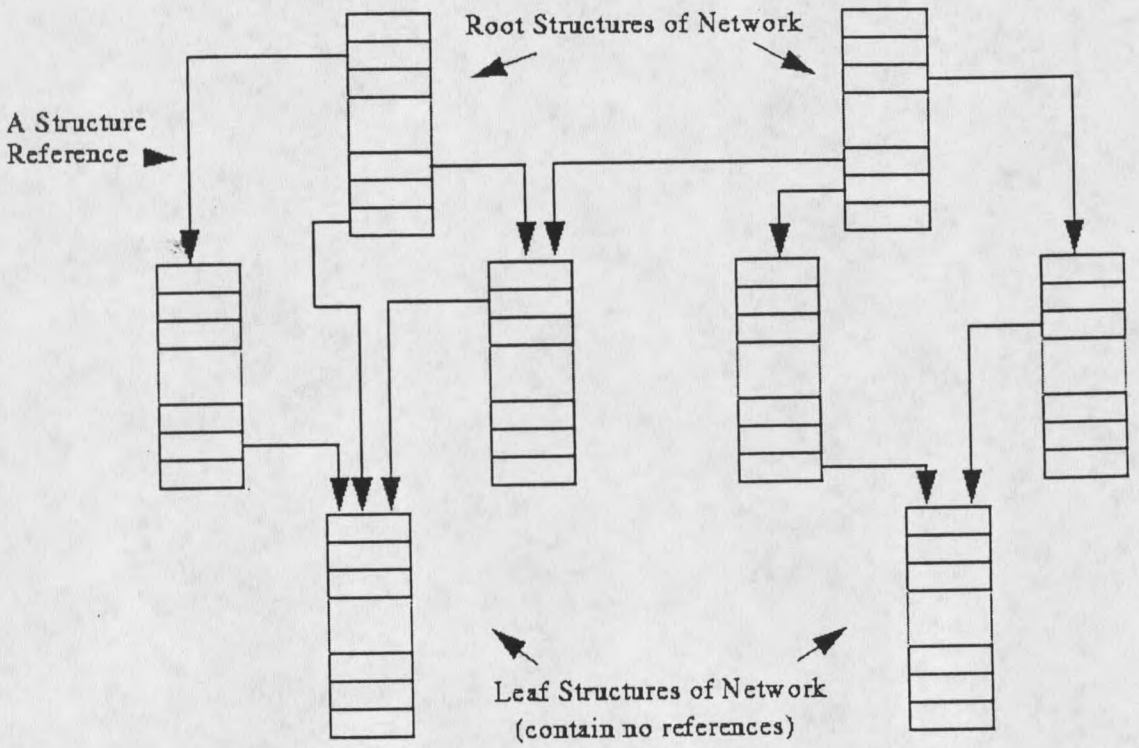


Figure 8. Structure Network

Rendering is accomplished through two parallel pipelines. The transformation pipeline handles the geometric transformations, from relating the model's components to each other, describing from where and how much of the model is viewed, to orienting the final image on the drawable. The rendering pipeline determines the graphic image's color, after applying numerous functionalities such as lighting, texturing and depth-cueing. The geometry transformation stages of the rendering pipeline is shown in Figure 9.

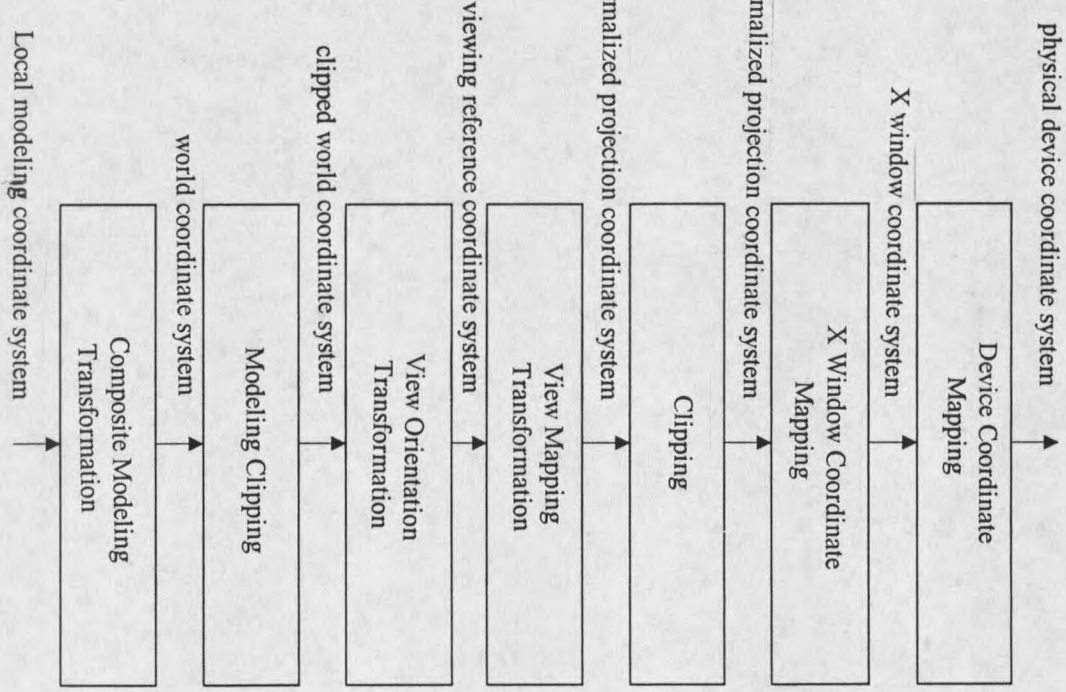


Figure 9. Geometry transformation stages of the rendering pipeline

The pick measure resource assists the PHIGS workstation resource in implementing PHIGS picking (hittesting) semantics. Clients are allowed to establish the parameters of the picking operation by modifying the initial state of a pick measure resource, and pick results are obtained by querying the attributes of the pick measure. Primitives are not rendered when the state is picking. The pick measure is only for backwards compatibility with PEX 5.1.

The PHIGS workstation is a PEX resource which encapsulates the PEX functionality to support the PHIGS abstraction of a workstation. The PHIGS definition of a workstation states that a workstation consists of zero or one display space. In X, many display spaces (windows) are possible in a single workstation display. The PHIGS workstation resource ties together a single X drawable, a set of lookup tables, highlighting and invisibility name sets, and all of the state need to make it perform as a PHIGS user expects a workstation should perform. The PHIGS workstation resource is maintained in PEX 5.2 and PEXlib 5.2 only for backwards-compatibility, because the original design goals of PEX have been changed. No new PEX 5.2 functionality is added to the workstation resource such as the PEX 5.2 texture mapping lookup table. However, some PEX 5.2 functionality may still be usable from the workstation subset, such as new output commands that do not implicitly reference new lookup tables or other new resources.

The PEX font manipulation mechanisms are the same mechanism as those for X11 fonts. As result, the PEX font resource is very similar to the font resource that is created and managed by X11. However, PEX fonts that are used with PEX stroke precision text contain more functionality than is currently found in X11 fonts. Specifically, stroke precision text requires text fonts that you can scale and rotate. Because of the added capabilities of PEX fonts, PEX has defined its own PEX font open, close and query requests. However, the PEXlib 5.2 functions correlate to the Xlib font functions, and PEX uses the X11 font path in order to locate PEX font files. Also, PEX 5.2 implementations are required to have a XLFD (X Logical Font Description) font name for each available font. These implementations may also provide alternate names for some fonts and for some backwards compatibility.

The Search Context is a PEX resource which performs a view-independent pick on a geometrical object which is defined by a structure network. It allows clients to establish the parameters for performing an incremental spatial search in world coordinates on output primitives stored in a structure hierarchy. Also, the pick is done with a structure network only.

Summary of the Architecture for PEXlib 5.2

PEXlib 5.2 uses the floating-point format native to the architecture of the machine (i.e., the machine on which the application executes). PEXlib 5.2 expects all of the floating-point values to be in that native format, with a few exceptions for functions which allow the application to specify the floating-point format. These functions, which enable the application to specify a floating-point format, deal primarily with application data that is already in protocol format, or data that the application wants to be encoded into protocol format for its own use. As a result, PEXlib 5.2 has a complex architecture which is used to support its large amount of functionalities. The architecture of PEXlib 5.2 leads the own characteristics that differ with other 3D API packages, such as the structure, database rendering and the relationship with X Window System.

The Architecture of OpenGL 1.0

OpenGL 1.0 draws primitives into a framebuffer subject to a number of selectable modes. Each primitive is a point, line segment, polygon, pixel rectangle, or bitmap. Each mode may be changed independently. Although many modes may interact to determine what eventually ends up in the

framebuffer, the setting of one does not affect the setting of others. Modes are set, primitives specified, and other OpenGL operations described by sending commands in the form of function or procedure calls. All elements of OpenGL architecture, even the contents of the texture memory and framebuffer, can be obtained by an OpenGL 1.0 application.

Architecture's Structure

The OpenGL 1.0 architecture is a good model for representing graphics problems. OpenGL 1.0 is carefully designed to support a broad array of advanced graphics rendering techniques, such as texture mapping, antialiasing, transparency and fog, as well as to achieve real-time rendering performance on accelerated graphics hardware. Figure 10 shows the architecture of OpenGL 1.0 and how OpenGL processes data. Commands enter OpenGL on the left. Most commands may be accumulated in a display list for processing at a later time. Otherwise, commands are effectively sent through a processing pipeline. The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values. The second stage operates on geometric primitives described by vertices: points, line segments and polygons. In this stage vertices are transformed and lit, and primitives are clipped to a viewing volume in

preparation for the next stage, rasterization. Finally, pixel rectangles and bitmaps bypass the vertex processing portion of the pipeline to send a block of fragments directly through rasterization to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer. Values may also be read back from the framebuffer or copied from one portion of framebuffer to another. These transfers may include some type of decoding or encoding operations.

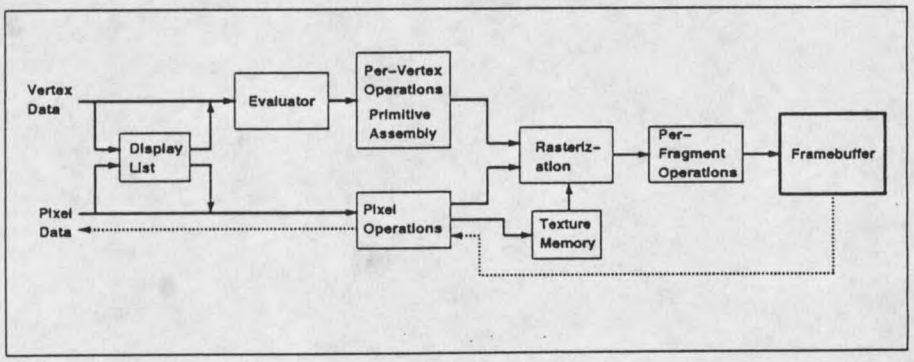


Figure 10. Architecture of OpenGL 1.0

The evaluator which can be found in Figure 10 , allows the specification of polynomial functions of one or two variables whose values determine primitives' vertex coordinates, normal coordinates, color, or texture coordinates. The evaluator interface provides a basis for building a more general curve and surface package on top of OpenGL. One advantage of providing the evaluator in OpenGL 1.0 instead of a more complex NURBS interface is that applications that represent curves and surfaces as other

than NURBS or that make use of special surface properties still have access to an efficient polynomial evaluator without incurring the cost of converting to a NURBS representation.

The rasterization is an important part in the architecture of OpenGL 1.0. It is the process by which a primitive is converted to 2D image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a color and a depth value to each such square.

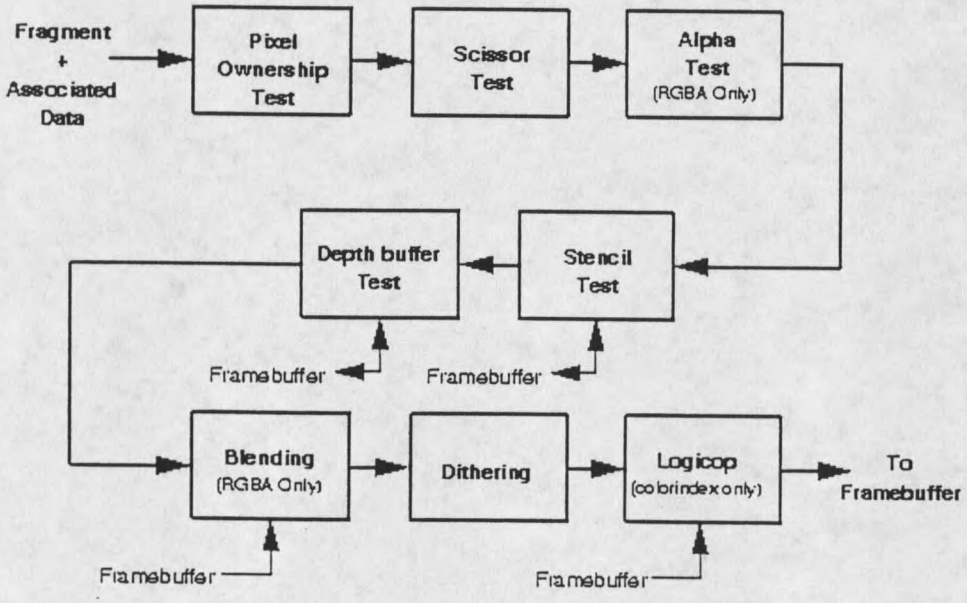


Figure 11. Per-Fragment Operations

The results of this process are passed on to the next stage — per-fragment operations which is shown in Figure 11, and it uses the information to update the appropriate locations in the framebuffer. In summary, the rasterizer produces a series of framebuffer addresses and values using a 2D description of a point, line segment or polygon. Each fragment so produced is fed to next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values, blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values.

Framebuffer

The destination of rasterized fragments is the framebuffer, where the results of OpenGL rendering may be displayed. In OpenGL 1.0, the framebuffer consists of a rectangular array of pixels corresponding to the window allocated for OpenGL rendering. Each pixel is simply a set of some number of bits. Corresponding bits from each pixel in the framebuffer are grouped together into a bitplane; each bitplane contains a single bit from each pixel. The bitplanes are grouped into several logical buffers: the

color, depth, stencil and accumulation buffers. The color buffer is where fragment color information is placed. Since OpenGL supports both double-buffering and stereo, the color buffer is further subdivided into four buffers: the front left & right buffers and the back left & right buffers. The front buffers are those that are typically displayed while the back buffers (in a double-buffered application) are being used to compose the next frame. The depth buffer is where fragment depth information is placed, and is typically used to effect hidden surface removal through Z-buffering. The stencil buffer contains values each of which may be updated whenever a corresponding fragment reaches the framebuffer. Stencil values are useful in multi-pass algorithms in which a scene is rendered several times. The accumulation buffer is also useful in multi-pass algorithms.

The model for interpretation of OpenGL 1.0 commands is client-server. That is, a program (the client) issues commands, and these commands are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client.

The effects of OpenGL 1.0 commands on the framebuffer are ultimately controlled by the window system which allocates framebuffer resources. It is the window system that determines which portions of the framebuffer OpenGL may access at any given time and that communicates to OpenGL how those portions are structured. Similarly, display of

framebuffer contents on a CRT monitor is not addressed by OpenGL. Framebuffer configuration occurs outside of OpenGL in conjunction with the window system; the initialization of an OpenGL context occurs when the window system allocates a window for OpenGL rendering. OpenGL 1.0 has no facilities for obtaining user input, since it is expected that any window system under which OpenGL 1.0 runs must already provide such facilities. These considerations make OpenGL 1.0 independent of any particular window system.

Immediate Mode in OpenGL 1.0

OpenGL 1.0 supports both immediate mode and display list mode, and they can be mixed within a single program. Although, SGI emphasizes its immediate mode approach to the graphics hardware/software interface [18], the immediate mode which OpenGL 1.0 provides is similar to PEXlib 5.2. With immediate mode the surface description is sent repeatedly, every frame, without retaining the previous surface description in graphics subsystem. This has the obvious cost of repeatedly sending data that may change infrequently. This is also a disadvantage for immediate mode. However, the immediate mode has two profound and subtle advantages.

First, immediate mode allows applications that display changing information to run as efficiently as those with

mostly static information. Immediate mode invites rather than punishes dynamic applications. Second, retained information in a graphics subsystem is essentially a "second database" that duplicates data structure inside the application [18]. When that data changes, the retained data must be edited synchronously. Because immediate mode doesn't need to synchronize an application's data structures with data stored in the graphics subsystem, it is far easier to program with static scenes.

Display List Mode

The display list mode is the distinctive architecture for OpenGL 1.0. While immediate mode provides flexibility, its use can be inefficient if unchanging parameters or objects must be respecified. To accommodate such situations, OpenGL provides display lists. A display list encapsulates a sequence of OpenGL 1.0 commands and is stored on the server. The display list is given a numeric name by the application when it is specified; the application need only name the display list to cause the server to effectively execute all the commands contained within the list. This mechanism provides a straightforward, effective means for an application to transmit a group of commands to the server just once even when those same commands must be executed many times. In a view with a display list, the CPU sends a

separate list of instructions that control the rendering or rasterization process to actually display a given view of an object. To change the properties of the geometric model, the user needs to go around the display list, usually by typing in new parameters for the model. The user can, of course, interactively move around a display list based scene just like an immediate mode scene [19].

Unlike PEX structures, the display lists are not editable. Once a display list is created, it is sealed and cannot be changed (except by destroying and recreating it). In other words, to optimize performance, a display list is a cache of command rather than a dynamic database [20]. This write only nature allows optimization to be performed on display lists unavailable to structures. The commands in the display list can be optimized for faster execution. An effect similar to display list editing may thus be obtained by: (1) building a list that invokes a number of subordinate lists; (2) redefining the subordinate lists. This redefinition is possible on a fine grain: a subordinate display list may contain anything, including just a single vertex or color command. There is no automatic saving or restoring of modes associated with display list execution[21].

Display lists are designed to optimize performance, particularly over networks, but never at the expense of performance on a local machine. Depending on this design

philosophy, display lists reside with the server and network traffic is minimized. Even locally, a display list might be more efficient since it can be processed as it's created into a form that's more compatible with the graphics hardware.

Display list mode provides a lot of confidence and power for users creating their own 3D graphic images. There are some advantages and disadvantages for it. One advantage of the display list is that a complex object needs to be described only once ever if it is to be displayed many times. Another advantage is just as same as for structures, the display list can minimize the amount of transfer overhead when running OpenGL 1.0 over a network since the commands in a display list can be executed repeatedly by only calling the display list by name. The commands themselves need to be transferred across the wire only once. The disadvantages of display lists are going with the problems of OpenGL. One disadvantage is that it can require considerable effort to re-specify the object if it is being continually modified as a result of user interaction. Another disadvantage of display lists is that display lists are difficult to execute in parallel, since the modes set in one display list must be in effect before a following display list is executed.

Summary the Architecture for OpenGL 1.0

Because of the architecture of OpenGL 1.0, it is not object-oriented and is not a descriptive graphics system[22]. It is a flexible procedural interface that allows users to describe a variety of 3D rendering tasks. It does not enforce a particular method of describing 3D objects, but instead provides the basic means by which those objects, no matter how they are described, may be rendered. Also, OpenGL 1.0 is rendering-only, so it is independent of the methods by which user input and other window system functions are achieved, marking the rendering portions of a graphical program that uses OpenGL platform-independent.

Analysis

After describing the architectures for PEXlib 5.2 and OpenGL 1.0, it is not difficult to see the commonality and differences between them. Both OpenGL 1.0 and PEXlib 5.2 requests can take substantial amounts of time to complete. Also, both OpenGL 1.0 and PEXlib 5.2 can easily generate requests with arbitrarily long execution time[23]. They have their own particularly features, such as structure for PEXlib 5.2 and display list for OpenGL 1.0.

In the case of PEXlib 5.2, PEX's architecture is very tightly coupled to the X Windows System. Not only was it

designed in the context of X but also its semantics depend on X notions of drawable, events and execution requirements. Of course, it inherited some good things from the X Windows System. However, OpenGL 1.0 separates the parts of the interface that are window-system dependent from those that are window-system independent. Adding a small new set of window-system dependent routines allows the remainder of OpenGL to be used without change. PEXlib 5.2 and OpenGL 1.0 have their own applications that fit user's requirements in the 3D graphics area. They will still grow up in their own ways and compete each other which will benefit the users in 3D graphics API market.

CHAPTER 4

COLORING

Today, computer graphics is becoming more and more complex and color plays an essential role. The purposes for using color in computer graphics are for aesthetics, to establish a tone or mood; for searching; for identifying areas as being associated; and for coding [24]. Users tend to like color, even when there is no quantitative evidence that it helps their performance.

Color Models

There are several color models available. They include: RGB (Red, Green and Black), CMY (Cyan, Magenta and Yellow), HSV (Hue, Saturation and Value), HLS (Hue, Lightness and Saturation), and CIE (Commission International del' Éclairage). The RGB and CMY models are hardware-oriented color models, while HSV is user-oriented.

RGB Model

RGB is the primary color type which is used by most of 3D graphics APIs because the RGB color type is used in CRT (Cathode Ray Tube) monitors and color raster graphics employs a Cartesian coordinate system. The RGB color model is shown in Figure 12. One corner — or vertex to be more precise — is black (0,0,0) and the opposite is white (1,1,1). From the black vertex, three edges go to the adjacent vertices: Red, Green and Blue. Red corresponds to the X axis, Green to the Y axis and Blue to the Z axis. At the three vertices adjacent to white are the mixtures of color. The mixture of green and red makes yellow (1,1,0); green and blue make cyan (0,1,1); red and blue make magenta (1,0,1). The gray scale runs through the middle of the cube from the black vertex to the white vertex. However, the problem with RGB is that it is not user friendly. So most of time, there are at least two color spaces provided by 3D graphics APIs.

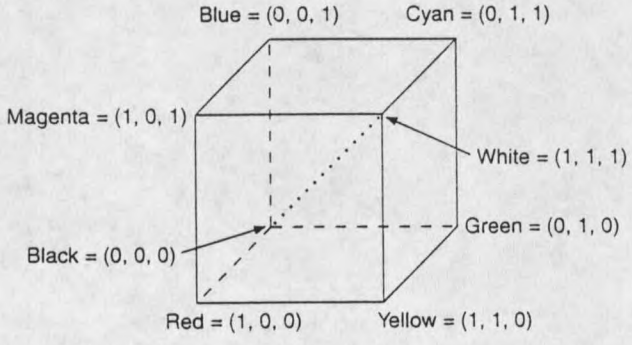


Figure 12. The RGB Cube

A computer graphics monitor emulates visible colors by lighting pixels with a combination of red, green and blue light in proportions that excite the red-sensitive, green-sensitive and blue-sensitive cones in the retina in such a way that it matches the excitation levels generated by the photon mix it's trying to emulate. To display a particular color, the monitor sends the right amounts of red, green and blue light to appropriately stimulate the different types of cone cells in human's eye. A color monitor can send different proportions of red, green and blue to each of the pixels, and the eye sees a million or so pinpoints of light, each with its own color.

Advantages For Using Color

Color offers a number of distinct advantages for display design. First are the obvious aesthetic benefits of color, supported by the general preference for color over monochromatic presentations. Second, color has the potential for greatly increasing information coding capability and flexibility and for reducing visual search time on complex display. Third is derived from addition of color contrast which can increase symbol visibility and reduce display brightness requirements [25].

Coloring in PEXlib 5.2

In PEXlib 5.2, a color is defined by color model and color value. The color model indicates the color spaces such as RGB; the color value indicates a color in that model. PEX gives the application total control over color throughout all phases of the graphics pipeline. Any color given to PEX may be a direct color type or an indexed color type (see Figure 13). A direct color can be specified by giving its color coordinates, and an indexed color can be specified by giving a color index. PEX uses this index to select a color from the renderer's color table. The colors in the color table are direct colors that have been previously stored there. Most of PEX 5.2 implementations support only RGB and indexed color.

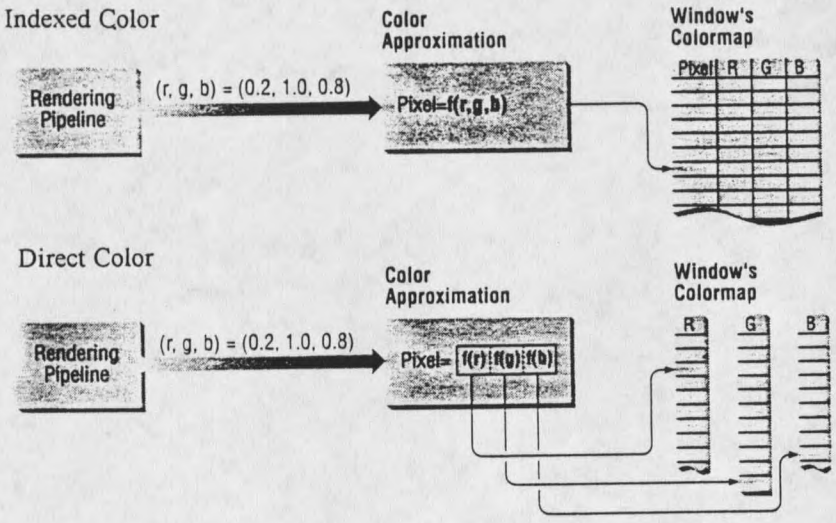


Figure 13. Direct Color and Indexed Color

Color Models in PEXlib 5.2

Although PEX 5.2 defines five color models which are RGB, RGBA, HSV, HLS and CIE, RGB is the only one generally available. Also, PEX 5.2 has four variants of the RGB color model: RGB8, RGBA8, RGB16 and RGBA16. These allow the color coordinates to be specified as integers instead of floating-point numbers. For the RGB8 and RGBA8 case, the color coordinates must lie between 0 and 255, with 255 being the maximum intensity. For RGB16 and RGBA16 the values must lie between 0 and 65535.

Note the distinction between a color model and a color type. A color model is a method for representing color, such as RGB, RGBA or HLS. However, a color type is an integer that indicates how a color is defined: indirectly or directly in some color model. For example, indexed color is not a color model. It is just a color type which is an indirect way to specify a color.

There are some important concepts along with PEXlib 5.2 color system, such as intrinsic color, color tables, color pipeline, indexed color, color approximation, etc.

Intrinsic Color

The intrinsic color is an important concept in the PEXlib 5.2 color system. As a primitive enters the rendering pipeline, its intrinsic color is determined. The intrinsic

color is the basic color of the primitive before it's lit, shaded and depth cued. It means the intrinsic color is the color that the object should be before any lighting, shading or depth cueing. The rule for determining the intrinsic color is a simple precedence rule: If vertex colors exist, then use them; if facet colors exist, then use them. If there is no color associated with the primitive, then use the surface color. This determination is only made for frontfacing surfaces. If a facet is backfacing and backface distinguish is on, then the backface surface color should be used.

Vertex colors have the highest precedence. Ignoring the shading method, the single facet color chosen should be some combination of the vertex colors. On the other hand, facet colors have the second highest precedence. Surface color will be used if there are no vertex or facet colors. In case of lines, text and markets, they do not participate in lighting or shading. Their intrinsic colors are provided by the primitive specific attributes and are only affected by depth cueing and color approximation which is used for shading and will be discussed later.

Color Pipeline

The color pipeline is another concept in the PEXlib 5.2 color system. It is an abstraction of color processing that

occurs in the graphics pipeline. The color pipeline is also referred to as the rendering pipeline. There are several stages to the color pipeline: intrinsic color resolution, lighting, depth cueing and color approximation. Figure 14 shows the stages of color pipeline [26].

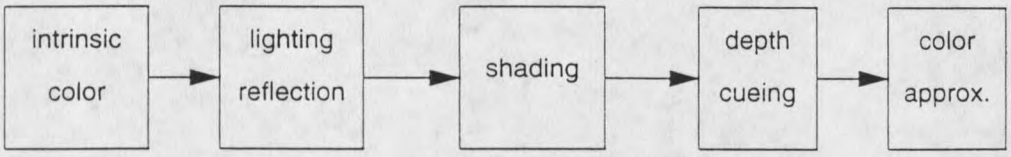


Figure 14. Stage of Color Pipeline

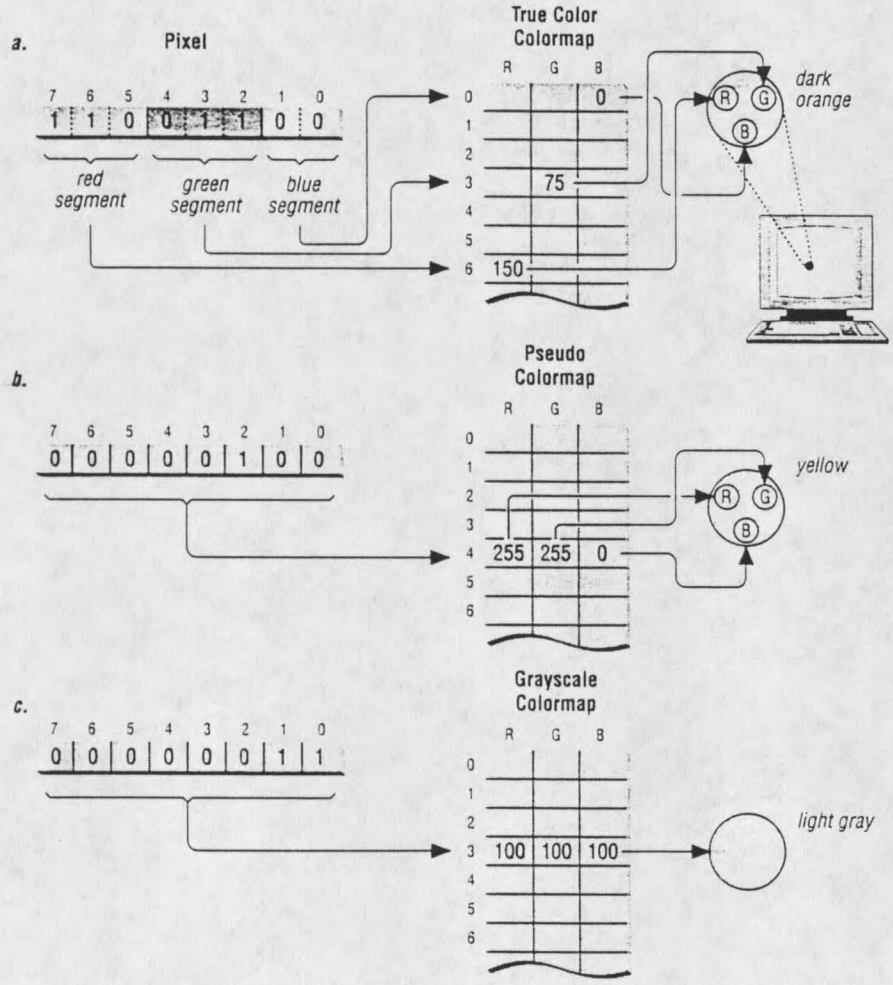


Figure 15. Processing for Directcolor, Pesudocolor and Grayscale visuals. a)Directcolor visuals b)Pesudocolor visuals c) Grayscale visuals.

Color approximation is the end of the color pipeline; the floating point RGB triplet used in the color pipeline gets turned into a pixel. The color approximation stage is always active, and all colors are subject to it. There are three ways for a pixel value to select a color from the window's color map.

Indexed Color

The indexed color which is an indirect color type is mainly used by PEXlib 5.2. It can be specified by indicating a color type of indexed and a color index. This index selects a color from a list of entries in a color table associated with the renderer. When drawing a primitive, PEX converts the index to a color by selecting the color table entry associated with that index, and using the color stored there. Figure 16 shows this conversion [28]. The index need not be defined in the color table when the attribute is set. If the index is not defined when the function is called in immediate mode, then entry 1 in the color table is used. If the attribute is being stored in a structure, then the color table entry will be defined later. If the entry is still not defined when the structure is traversed, PEX uses color table entry 1. It means all of the

indexed colors will resolve to white, which will make for a boring picture.

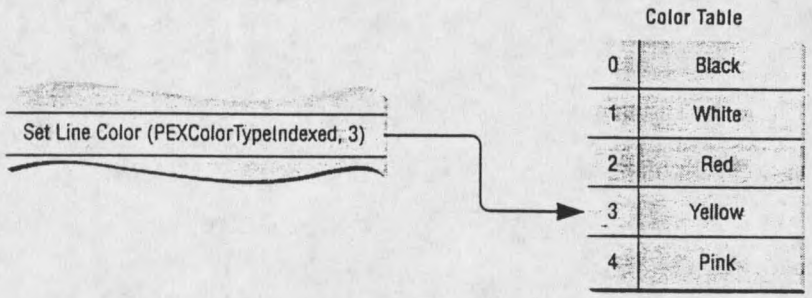


Figure 16. PEX resolves an indexed color by selecting the color table entry for the index

There are some advantages for using indexed color. One is that changing colors can be easily performed. Another advantage is that indexed color may save memory in structures because color index structure elements typically use one-fourth the space of color elements. One of the disadvantages for indexed color is that it is not easily to be understood by some users, since it has more complicated structures than the direct color.

Each method of color specification has its own advantages and disadvantages. Which to use depends on the needs of the application. Neither direct color nor indexed color can be used always. They can be mixed in the same application.

Color Table

The last concept for PEXlib 5.2 color system is color table. The color table, which is shown in Figure 17 with some entries and all in the RGB model, is a list of colors with an associated indexes. Index values start at zero, which is the first entry in the table. The colors in the table can be in different color models, but they typically are not. There are typically some predefined entries in the color table. Those entries do not need to be contiguous. The number of predefined entries and the colors they define vary among PEX 5.2 implementations.

Color Index	Color model	Color value
0	RGB	0.0, 0.0, 0.0
1	RGB	1.0, 0.0, 0.0
2	RGB	0.0, 1.0, 0.0
3	RGB	0.0, 0.0, 0.0
4	RGB	0.3, 1.0, 0.5
5	RGB	0.1, 0.6, 0.9

Figure 17. A color table

Each PEX renderer may have its own color table. The color table of a renderer is not the hardware color table of the computer. The colors in the table may not even be exactly displayable by the window, and may therefore be approximated by colors that can be displayed. The color

tables of two separate renderers can be different, this allows that the same color index in an output command may produce different colors when interpreted by different renderers.

Because color tables are an instance of a PEX 5.2 lookup table, it is created the same way all lookup tables are created.

Summary of the Coloring System in PEXlib 5.2

PEXlib 5.2 provides a very nice coloring system which gives a lot of useful features for users to create their own color system. It has several color models and color types for users to choose and mix together to fit their applications. These color models and color types are well designed and organized for applications to choose. Also, the coloring system for PEXlib 5.2 is fast, simple and efficient.

Coloring in OpenGL 1.0

Since PEXlib was published early the OpenGL, the color system for OpenGL 1.0 inherited some advantages from PEXlib. They have similar color systems.

The goal of almost all OpenGL applications is to draw color pictures in a window on the screen. The window is a

rectangular array of pixels, each of which contains and displays its own color. Thus, in a sense, the point of all the calculations performed by an OpenGL 1.0 implementation — calculations that take into account OpenGL 1.0 commands, state information and values of parameters, is to determine the final color of every pixel that's to be drawn in the window [29]. For more details about how a pixel can be drawn in a window with colors see the OpenGL Programming Guide. OpenGL 1.0 provides two display color modes: RGBA and Color-Index mode. The basic ideas for them are as same as those in PEXlib 5.2.

Color Modes

Because OpenGL 1.0 is a window independent system, the color mode must be set up with window system's initialization. The color map becomes a window system operation when color index mode is chosen. Once the color display mode is initialized, it can't be changed. As the program executes, a color that either a color index or an RGBA value is determined on a per-vertex basis for each geometric primitive. The processing of colors in OpenGL 1.0 is shown in Figure 18.

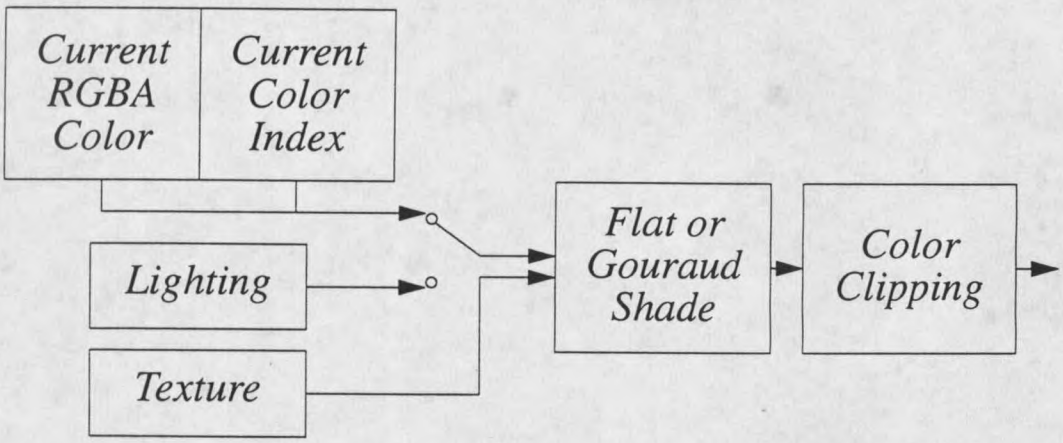


Figure 18. Processing of Colors in OpenGL 1.0

In both color-index and RGBA modes, a certain amount of color data is stored at each pixel. This amount is determined by the number of bitplanes in the framebuffer. A bitplane contains one bit of data for each pixel. There are 8 color bits per pixel and hence 256 different values or colors that can be stored at the pixel. Bitplanes are often divided faintly evenly into storage for R, G and B components, but this isn't always true. OpenGL provides function calls to help users find out the number of bitplanes available on their system.

RGBA Mode

In RGBA mode, the hardware sets aside a certain number of bitplanes for each of the R, G, B and A components. The R, G and B values are typically stored as integers rather than floating point numbers, and they are scaled to the

number of available bits for storage and retrieval, which is shown in Figure 19 [30].

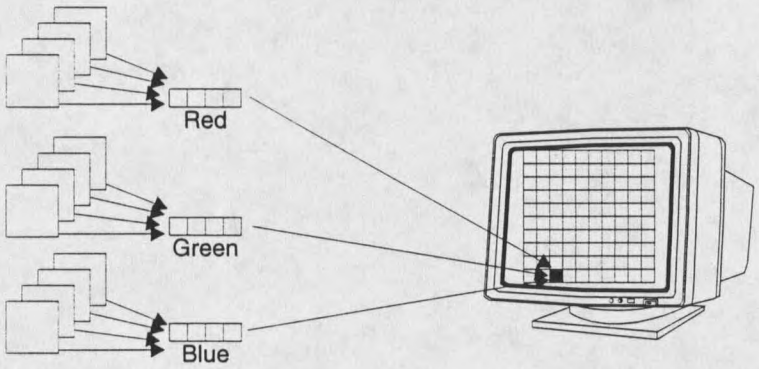


Figure 19. RGB Values from the Bitplanes

The A in RGBA means the alpha value. It has no direct effect on the color display on the screen. It can be used for many things, including blending and transparency, and it can have an effect on the values of R, G and B. The alpha value is between 0.0 which is the minimum intensity and 1.0 which specifies the maximum intensity, and it isn't specified in color index mode. The alpha component is never displayed directly. It's typically used to control color blending. By convention, OpenGL alpha corresponds to the notion of opacity rather than transparency, meaning that an alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 implies complete transparency. As a result, the blending operations are not performed in color-index mode. Also, each pixel's color is independent of other pixels with RGBA mode.

Color-Index Mode

In color-index mode, a color map is used. The idea for color map is same as that in PEX. Also, it is similar to use a palette to mix paints to prepare for a paint-by-number scene. It provides indices where the primary red, green and blue values can be mixed. The system stores the color index in the bitplanes for each pixel. Then those bitplane values reference the color map, and the screen is painted with the corresponding red, green and blue values from the color map. Each pixel with the same index stored in its bitplanes shares the same color map location. If the contents of an entry in the color map change, then all pixels of that color index change their color. The number of simultaneously available colors is limited by the size of the color map and the number of bitplanes available. The size of the color map is determined by the amount of hardware dedicated to it. Generally, sizes range is from 256 (2^8) to 4096 (2^{16}). The size of the color map is a power of 2, indexed by the number of bitplanes available in color index mode.

For both color-index and RGBA mode, they have their application areas for users. Also, they both have some advantages and disadvantages. With RGBA mode, for several effects such as shading, lighting, texture mapping and fog, RGBA provides more flexibility than color-index mode. It is based on what hardware is available for choosing the color

display mode (RGBA or color index mode). In most system, RGBA mode can simultaneously represent more colors than color index mode. On the other hand, color-index mode can be useful for various tricks, such as color map animation and drawing in layers. It also might work better if there are limited shading requirements, such as only shades of gray. When the numbers of bitplanes are small, color-index mode should be considered.

Analysis

Because the color system is an important part for 3D APIs, PEXlib 5.2 and OpenGL 1.0 both provide nice color systems to fit their architectures. By contrast, PEXlib 5.2 has a richer and more flexible color system than OpenGL. The main reason is that PEXlib was designed with the X Window System, so it is easy to provide more color systems with the X Window System. However, OpenGL is designed for window system independent. It is hard to provide a lot of color systems to fit different window systems. The color system which is used in OpenGL 1.0, is the best to fit the hardware so that the property of window independence can be easily realized. PEXlib 5.2's color system is complex and powerful, while OpenGL 1.0's color system is simple and practical. They both are fast systems and have a lot of similar points.

CHAPTER 5

LIGHTING

In creating a 3D environment it is very important to add light sources and shading effects to enhance the 3D graphics effect. If we assume that there is no light at all in an environment, for example, a windowless room, with the lights turned off — nothing can be seen. The rendering of a 3D computer graphics scene is similar. In order to "see" — that is, to render — a scene or any object within that scene, light must be defined.

Lighting is the process of calculating how much light reaches an object and usually refers to the calculation of how much light is reflected off of the object. This is referred to as the reflection and the reflectance calculation which will be discussed later.

Basic Principles of Lighting

When defining lights with a 3D graphic package, such as PEXlib 5.2 and OpenGL 1.0, there are three basic characteristics that lights process in the real world that must be defined. The first is location, or where the lights are in the 3D scene. Another parameter to be defined for all lights is intensity, which usually is specified by a range of numbers. The higher the number, the brighter the light. The third parameter to be specified is the color of the light. Usually, the color of the light is defined by three numbers. These numbers are referred to as RGB values.

In addition to determining the location, the intensity and the color of a light, the type of light must be defined. Although there are many different types of light in the real world, such as fluorescent lights, incandescent light (the glow of standard light bulbs), daylight and candlelight etc, most 3D graphics only offer a range of light types that attempts to simulate the different types of real light.

Types of Light

There are five types of light commonly used by 3D graphics systems. They are ambient light, directional light

(infinite light), point light, spotlight (specular light) and area light, which are shown in Figure 20.

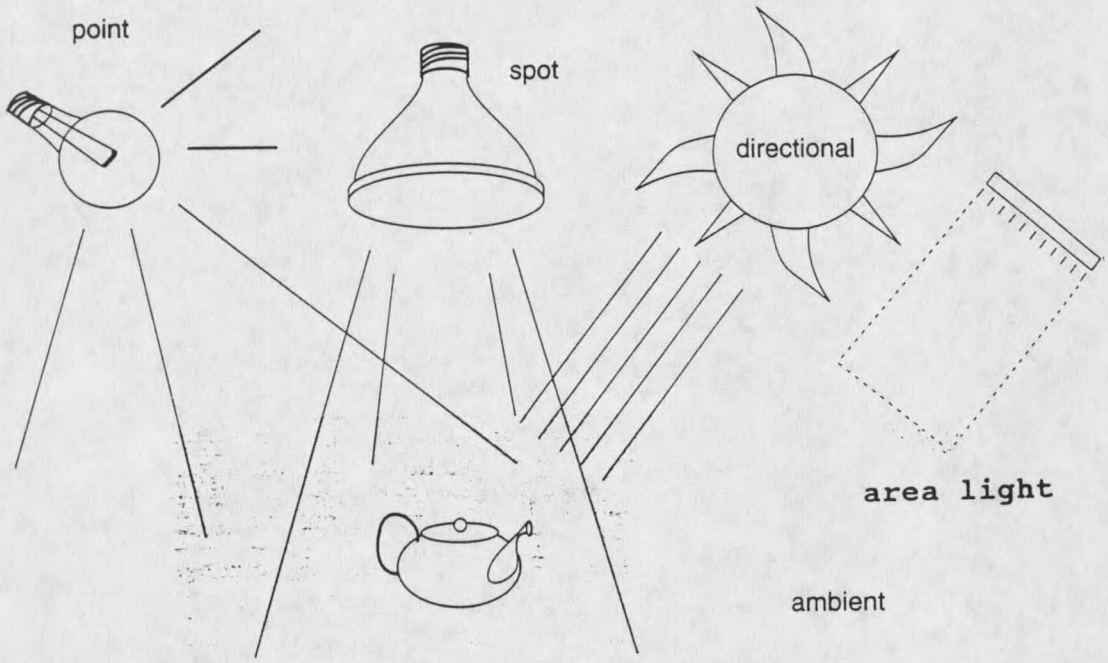


Figure 20. Five Light Types

Ambient light simulates the overall amount of daylight (or lack of daylight) in the world around you. An ambient light does not emanate from a specific location in space, nor does it flow in a specific direction. It is simply an overall level of light that is presented everywhere in the environment. Consequently, when defining an ambient light, only the intensity and color parameter have to be specified. Also, it is useful to identify lighting problems with your program because reducing the ambient light to black is helpful to eliminate the color shifting effect of the ambient light. An ambient light must be used in conjunction

with other lights since it can not provide a realistic rendering of a 3D situation by itself [31].

The directional light is considered to be located infinitely far away (simulating, in effect, the great distance of the sun). In this it resembles ambient light, which has no location. However, directional light, unlike an ambient light, has directionality, in that it moves from right to left, or from upper left to lower right, and so on. In this, it specifies the sunlight more accurately which comes from a certain direction. However, unlike other lights, which also have directionality but that throw light within only circumscribed regions, such as spotlights and area lights, directional light throws equal amounts of light everywhere. The light comes from a specific direction, but it comes from that direction equally throughout the scene. One advantage of directional lights is that it is very easy to get a reasonable rendering with only a single directional light. Another advantage is that it is very easy and not very time-consuming for the computer to calculate the effects of a directional light.

The point light is the most common type of light in computer graphics systems [32]. The point light simulates the effect of a bare light bulb hanging from a wire and located at a specific point in space. It radiates light equally in all directions. There are three basic parameters that used to define the point light. They are: location,

color and intensity. In addition to these parameters, the decay parameter is also important for a point light. The decay parameter is designed to simulate the fact that light from a light bulb does not travel forever, and fades with distance. A decay value of zero means that there is no decay. In other words, the light continues forever at the same intensity. It is a very unrealistic situation. If the decay value is larger, it means that the light fades more quickly as it moves away from the point light. Unlike a real light bulb, however, a point light does not have any size or any shape. It has no geometry. It is formless.

The spotlight is the second most common type of light source which is used by most of computer graphics systems. It is located at a specific point in space and has intensity, color and decay parameters. These parameters can be specified in the same way that they are specified for a point light. In addition to these parameters, however, a spotlight has several additional parameters which simulate the effect of a spotlight in the physical world. For example, the spread, which controls the width of a cone, is the measure of the angle between the opposite sides of the cone. Objects that lie outside of the cone do not receive any of the light from the spotlight. A spotlight also throws light in a certain direction. The cone of a spotlight can not be seen. Spotlights fade with distance because the intensity of light fades not only with distance from the

light source, but also with distance from the center of the light cone. That is, the intensity of the spotlight decreases from the center of the light cone to the edges of the light cone. The spotlight is useful for simulating the flush light and torch effects.

The last type of light is area light. The area light simulates the effect of an entire area emitting light rather than a single point. It begins at a rectangular area and emanates in a rectangular beam. The area light is useful to simulate the bank of fluorescent lights in the ceilings of some offices, and the screen of television set.

How to use light types efficiently and how to define them easily has become more and more important for 3D graphic packages.

Lighting in PEXlib 5.2

Types of Light Sources in PEXlib 5.2

Lights in PEX 5.2 provide a way to model some of the most common light sources in real world. They are: point light, spotlight, directional light and ambient light. Figure 21 shows them and the table 2 lists the light types defined by PEX 5.2. Users should not be under the illusion that any one of them will accurately capture the

interactions of light and shadow that occur in any nontrivial scene.

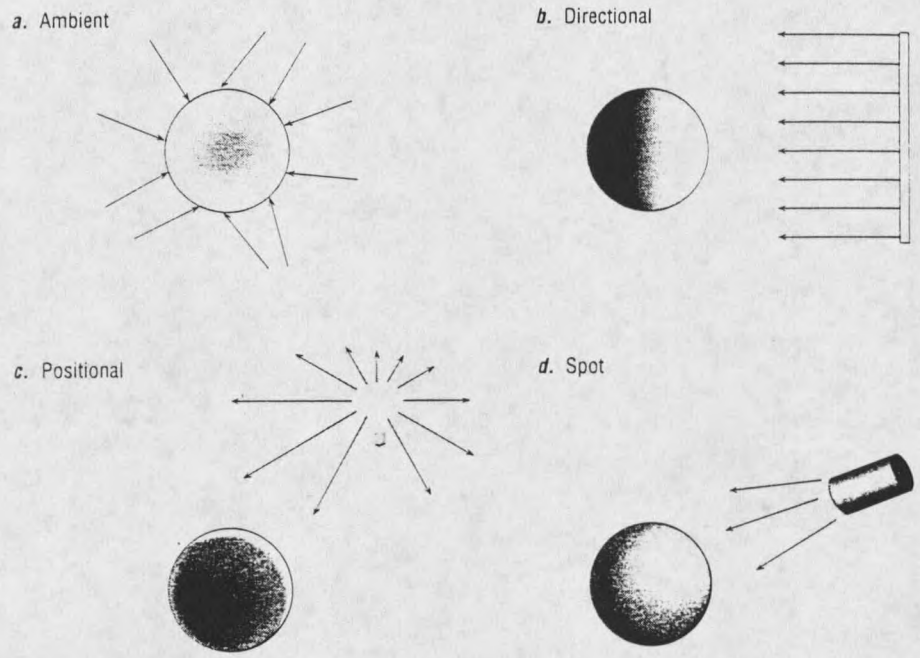


Figure 21. The PEX 5.2 Light Sources

Table 2. PEX 5.2 Light Types

Mnemonic	Light Type	PEXLightEntry Members Used
PEXLightAmbient	Ambient	type, color
PEXLightWCVector	Directional	type, color, direction
PEXLightWCPiont	Point	type, color, point, attenuation1, attenuation2
PEXLightWCSpot	Spot	type, color, point, direction, attenuation1, attenuation2, concentration, spread angle

In PEXlib 5.2, lights are defined as entries in the Light Table which is an attribute of the renderer. Lights

are enabled to be turned on by the presence of their table index in the Light State PC attribute and can be turned on and off using the PEXSetLightSourceState() function call during rendering. Also, the current values of lights can be inquired in the light table by using the lookup table functions[33].

As we mentioned before, ambient light is all around. It has no direction and does not vary in intensity. The only property of an ambient light is its color, so that is the only value needed to set an ambient light in PEXlib 5.2. Since ambient light is just a color, any number of ambient lights can be combined into one, and if more than one is specified the server usually will combine them in one. The amount of available light from all of the ambient light sources will be the sum of the light colors. By experience, it is good to have some ambient light in the scenes that you are designing so the unlit sides of objects are still visible. It is also helpful for adjusting other types of light in the scenes in PEXlib 5.2 programming.

The concept of directional light in PEXlib 5.2 is the same as what we described. The directional light is named WCVector to simulate sunlight in PEX 5.2. The directional light is specified by a color and the direction vector. The direction vector points in the direction the photons travel from the light to the object. The amount of available light

will be all of the light color for a directional light. It does not vary with the position of the object.

The point light is very useful. It emits light in all direction from a location in world coordinates. It is called WCPoint in PEXlib 5.2 call. It is defined by its color, location and three attenuation coefficients in PEXlib 5.2. The three attenuation coefficients indicate how the light's intensity diminishes with distance from the light source. The attenuation coefficients are used within the lighting calculation to determine how strong the light is when it reaches a primitive. The formula PEX 5.2 uses to compute the light attenuation is:

$$L_a = 1 \; / \; (C_1 + C_2|O_p - L_p| + C_3|O_p - L_p|^2)$$

Here L_a is the light attenuation factor which is multiplied against the light color during the lighting calculation[34]. If L_a is less than 1, the light is attenuated. Otherwise, if L_a is greater than 1, then the light is amplified which is an unnatural effect that is not often used. For more details about this formula see the PEXlib Specification and C Language Binding (Version 5.2). Finally, the available light from a point light will be the light color times the result of the attenuation calculation which also include the location factor.

Spot lights use every field in the light table entry to create a cone of light that diminishes with both distance from the light source and angle away from the center of the

cone. Objects outside the cone are not affected by the light at all. Figure 22 shows the meaning of parameters for spotlight sources. The whole light table entry's data structure is as follow:

```
typedef struct {
      PEXEnumTypeindex        type;
      PEXVector               direction;
      PEXCoord                point;
      float                   concentration;
      float                   spread_angle;
      float                   attenuation 1;
      float                   attenuation 2;
      float                   attenuation 3;
      PEXColorSpecifier       color;
      } PEXLightEntry;
```

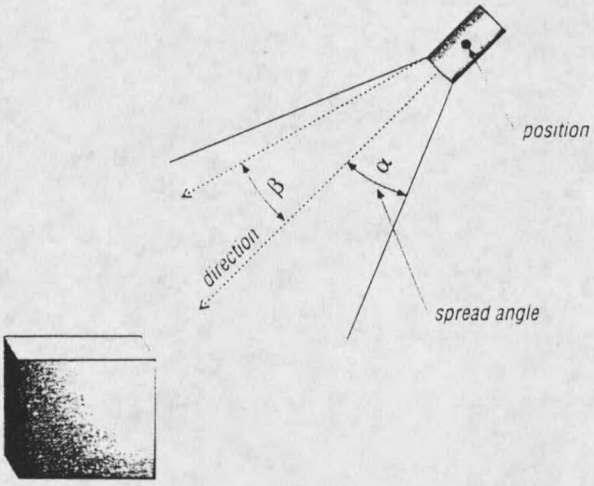


Figure 22. The Parameters of A Spotlight

The center of the cone is specified with the direction field, the point of the cone with the point field, the attenuation {1,2,3} fields are exactly the same as with the point light, and color is always the same for lights. It means that the color, location and attenuation coefficients serve the same purpose as the identical parameters of point

light sources. The location is in world coordinates and direction too. However, the spread_angle and concentration fields are used to control the beam of light. The spread angle denotes the width of the light cone by indicating its included half-angle, in radians. Points outside this cone are not struck by light from the light source. The spread angle can be no more than π radians; and some implementations calculate the light's effect incorrectly if the angle is greater than $\pi/2$. The concentration is used as an exponent of the dot product of the vector from the light to the object, and the direction of the light. It gives a measure of how concentrated the beam is. A low concentration will model a flood light and a high concentration will produce a focused beam of light. A very large concentration might simulate a laser. A zero concentration makes the beam a constant intensity throughout the cone. When inside the cone of a spotlight, the available light will be the light color scaled by the distance attenuation and the concentration. Since the spotlight is a complex concept, for more details see PEXlib Programming Manual and PEXlib Specification and C Language Binding (Version 5.2).

Once lights are defined, the next important step is to turn them on. Lights are turned on and off by the function PEXSetLightSourceState(). This function takes two lists: the activation list and the deactivation list. The activation

list indicates which lights to turn on. Otherwise, the deactivation list indicates which lights to turn off. These lists contain the indices of light source definitions in the renderer's light source table which is a list of active lights. Also, the two lists passed to PEXSetLightSourceState enable structures within a structure network to turn lights on or off. Turning a light on or off in one structure, however, does not prevent a structure lower in the network from turning it back on.

Reflection

In the previous section, the light is determined by the amount of light coming from some number of the four different light types. One thing is remaining. It is the concept of reflection which specifies the amount of light bouncing off a surface and the type of the surface. How important is the reflection? Let's imagine, if the photons did not bounce off the geometry, turning on lights would be very boring. The scene will be difficult to accept. I.e., when determining how much light is striking a surface, it has to be determined that how much light should bounce off the surface.

PEX 5.2 defines three types of reflection. They are: Ambient reflection, Diffuse reflection and Specular reflection. These three reflection types are shown in Table

3. There are two pieces of the pipeline contexts state that control the reflection of light off surfaces. The two are the reflection attributes and reflection type. For each light, the colors calculated for the ambient reflection, the diffuse reflection, and the specular reflection are added together become the final calculation of lighting and reflection.

Table 3. Three Types of Reflection in PEX 5.2

Reflection Type	**Incident Light**	**Reflected Light**
Ambient	Non-directed	Non-directed
Diffuse	Directed	Non-directed
Specular	Directed	Directed

Ambient reflection reflects only ambient light. It is as through the light comes from the object itself, or an extremely diffuse light source. Since ambient light comes from all directions, ambient reflection is reflected in all directions. Also, ambient light is reflected uniformly and with the same intensity from all points on an object's surface, regardless of the position of the object or the orientation or curvature of its surface. The ambient reflection does not really match a physical phenomena but is useful for lighting the dark side of objects.

Diffuse reflection redistributes nonambient light in all directions. The amount of light that bounces off the

surface depends on the orientation of the surface with respect to the light, the diffuse reflection coefficient and the amount of nonambient light available. The position of the viewer does not affect the amount of diffuse reflection[35]. The intensity of diffuse reflections varies in proportion to the angle at which the light strikes the surface. Thus, the intensity of diffuse reflections caused by a particular light source depends on the direction of light emitted by that source. Also, the orientation of the surface is determined by the vertex normals on a per vertex basis. The amount of light reflected is calculated by the dot product of the vertex normal and the normal to the light, or the inverse of the direction vector for directional lights, diffuse reflection coefficient, the amount of available light, and the intrinsic or object color.

Unlike the ambient reflection and the diffuse reflection, the specular reflection is more complex than them. Specular reflection produces specular highlights. These are the shiny spots on a smooth surface that reflect the light source directly. The important thing is that these highlights only occur when the angle to the viewer is equal to the angle of reflection. It is the angle called as mirror angle in PEXlib 5.2. The specular color is computed from the specular coefficient, the available light, the object's specular color and the dot product of the reflection vector and the

vector to the viewer raised to the specular exponent. In reality, specular reflections are indeed affected by the object to one degree or another. The surface may be slightly rough, breaking up the light just a bit and reflecting it in slightly different directions. The highlights still can be seen even if they are more diffuse and not as bright. Also, a specular reflection might carry some of the object's color because the object may absorb some of the rays and reflect only a portion of those that strike it.

So far, there are four types of light and three types of reflection. The Table 4 shows the relationships between them. Also, there are six reflection attributes defined by PEX. They are ambient reflection coefficient, diffuse reflection coefficient, transmission factor, specular reflection coefficient, specular color and specular concentration. The last three have to do with specular reflections.

Table 4. The Relationships Between the Light Types and the Reflection Types in PEXlib 5.2

	Ambient Reflection	Diffuse Reflection	Specular Reflection
Point Light	N	Y	Y
Spot Light	N	Y	Y
Directional Light	N	Y	Y
Ambient Light	Y	N	N

Reflection attributes are set all at once with a set reflection attribute output commands that is created by calling PEXSetReflectionAttributes. When traversed, the output command sets the reflection attributes in the pipeline state list. All primitives subsequently displayed use those attributes. To determine reflected color, PEX considers the geometry of the primitive, its color and reflection attributes and the light sources that shine on it. For more details about these reflection attributes, see the PEXlib Programming Manual.

Summary of the Lighting System in PEXlib 5.2

PEXlib 5.2 provides the lighting system as completely as it can, because the goal for the PEXlib 5.2 lighting system is to provide as many functions as it can to allow users to simulate lighting effects in the real world by using computers. From the view of users, the PEXlib 5.2 lighting system is good enough for users to simulate the real world lighting effects. Although there are a lot of equations and formulas behind the lighting system in PEXlib 5.2, the PEXlib 5.2 lighting system still runs well and it efficiently presents the scene which is well defined by users to simulate the real world. The PEXlib 5.2 lighting system is perfectly designed and easy to use. It constitutes an indispensable part of PEXlib 5.2.

Lighting in OpenGL 1.0

Types of Light Sources in OpenGL 1.0

OpenGL 1.0 provides four types of light sources: ambient, diffuse, specular and position. In OpenGL 1.0, the lighting model considers the lighting to be divided into four independent components: emitted, diffuse, ambient and specular, and all four components are computed independently and then added together. OpenGL 1.0 provides almost the same light sources as PEXlib 5.2. The diffuse light is similar to the directional light, and the specular light is similar to the spot light in PEXlib 5.2. The emitted light is the light that originates from an object and is unaffected by any light sources. However, the light sources have an effect only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material might emit its own light (TV screen), it might scatter some incoming light in all directions, and it might reflect some portion of the incoming light in a preferential direction like a mirror or shiny surface. OpenGL 1.0 provides three types of reflectances for materials: ambient, diffuse and specular reflectances. Also, there are a number of properties in

OpenGL 1.0 to define a light, such as color which is in the RGBA model, position and direction.

To specify a light model, there are three things that have to be defined before turning lights on or off. They are: how many light sources, the parameters defining the characteristics of these light sources and current materials. Since a light model depends on parameters, lighting parameters can be divided into three categories: light source parameters, light type parameters and material parameters. Table 5 gives each of the three parameter groups and the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each[36]. One light has a set of light parameters. At least eight lights can be defined in the scene. It is possible to have more, but it depends on the OpenGL 1.0 implementation. Since OpenGL 1.0 needs to perform calculations to determine how much light each vertex receives from each light source, increasing the number of lights adversely effects the performance.

Table 5. Correspondence of Lighting Parameter Symbols to Names. AMBIENT_AND_DIFFUSE Is Used to Set a_{cm} and d_{cm} to the Same Value.

Parameter	Name	Number of Values
Material Parameters (**Material**)		
a_{cm}	AMBIENT	4
d_{cm}	DIFFUSE	4
a_{cm}, d_{cm}	AMBIENT_AND_DIFFUSE	4
s_{cm}	SPECULAR	4
e_{cm}	EMISSION	4
s_{rm}	SHININESS	1
a_m, d_m, s_m	COLOR_INDEXES	3
Light Source Parameters (**Light**)		
a_{cli}	AMBIENT	4
d_{cli}	DIFFUSE	4
s_{di}	SPECULAR	4
P_{pli}	POSITION	4
s_{dli}	SPOT_DIRECTION	3
s_{rli}	SPOT_EXPONENT	1
c_{rli}	SPOT_CUTOFF	1
k_0	CONSTANT_ATTENUATION	1
k_1	LINEAR_ATTENUATION	1
k_2	QUADRATIC_ATTENUATION	1
Lighting Model Parameters (**LightModel**)		
a_{cs}	LIGHT_MODEL_AMBIENT	4
v_{bs}	LIGHT_MODEL_LOCAL_VIEWER	1
t_{bs}	LIGHT_MODEL_TWO_SIDE	1

Lighting Properties

Lighting properties defined by OpenGL 1.0 are clearly described in Table 6. Most of parameters have the same meaning as those in PEXlib 5.2. Of course, there are some new parameters only used for OpenGL 1.0, such as shininess, emission, global ambient and two-sided lighting. Note that the ambient, diffuse and specular contributions are all attenuated. Only the emission and global ambient values are not attenuated. The attenuation factor has the same meaning

and formula as that in PEXlib 5.2. Since the specular reflection from an object produces highlights, the amount of specular reflection seen by a viewer does depend on the location of the viewpoint. The shininess parameter is used with specular highlight to control the size and brightness of the highlight. The higher value for shininess, the smaller and brighter highlight. Otherwise, the lower value for shininess, the highlight is larger and darker. For more details about shininess, see the OpenGL Programming Guide on page 188.

The emission parameter is used to describe the material properties. It is used to simulate lamps, TV screen and CRT. This feature does not exist in PEXlib 5.2.

Table 6. Lighting Properties Defined by OpenGL 1.0

Parameter Group	Parameters
Material	diffuse specular and shininess ambient emission
Light Sources	color position local or infinite attenuation (drop off) spot (directional)
Lighting Model	global ambient local or infinite viewer two-sided lighting

As discussed earlier, each light source can contribute ambient light to a scene. In addition, there can be other

ambient light that is not from any particular source. To specify the RGBA intensity of such global ambient light, OpenGL 1.0 uses the global ambient light parameter.

Lighting calculations are performed for all polygons, whether they are front-facing or back-facing. Since the lighting conditions are usually set up with the front-facing polygons, the back-facing ones typically are not correctly lit. If the user does not care what the back-facing polygons look like, polygons is fine to use only lighting effect on the front-facing. However, if the user cares about the back-facing lighting effect, at this time the two-sided lighting parameter will take care of the back-facing and front-facing lighting.

In OpenGL 1.0, the spotlight source can be realized by restricting the shape of the positional light source. OpenGL provides some parameters which the user can use to provide a real spotlight, such as spot_direction, spot_exponent and spot_cutoff. For more details, see the OpenGL Programming Guide on page 166.

Lighting in Color-index Mode

Lighting in color-index mode has some differences. In color-index mode, the parameters comprising RGBA values either have no effect or have an interpretation. Since it is much harder to achieve certain effects in color-index mode,

users should use RGBA whenever possible. Also, because the allowable parameters for color-index mode are different from those for RGBA mode, the calculations for lighting effects are different as well. There is no material emission and no ambient light, the only terms of interest from RGBA equations are the diffuse and specular contributions from the light source and the shininess. However, these need to be modified to use different formulas to obtain lighting effects. For details, see OpenGL Programming Guide on page 192.

Summary of the Lighting System in OpenGL 1.0

The OpenGL 1.0 lighting system not only inherited some good ideas from the PEXlib 5.2 lighting system, but also improved some things in some ways, such as emission and shininess. PEXlib 5.2 uses RGB mode for the color of light, and OpenGL 1.0 uses RGBA model for the color of light. PEXlib 5.2 uses world coordinates to define the location of the light sources, and OpenGL 1.0 uses local coordinates.

Analysis

The PEXlib 5.2 and the OpenGL 1.0 lighting systems both provide good light systems and both have their own styles. Since both PEXlib 5.2 and OpenGL 1.0 provide ambient,

diffuse and specular light reflections, and combine them together for final lighting reflection effects, it means that both of them use and support the Phong light model [37]. The Phong light model is a classic local reflection model and it is the most commonly used lighting model in computer graphics [38]. Phong light model is a popular and efficient method to calculate the light reflection effect. In the PEXlib 5.2 lighting system, there are a lot of parameters that the user must define. Sometimes it is good for programmers. However, most new PEXlib programmers have a difficult time dealing with them. In the other hand, there are less parameters in OpenGL 1.0 lighting system than in the PEXlib 5.2 system. The parameters are easy to understand in OpenGL 1.0. In short, OpenGL 1.0's lighting system provides an easy and efficient way to simulate the lighting effects, while PEXlib 5.2's lighting system provides a rich and expensive way to simulate the lighting effect. They both can reach almost the same effects and implementations for lighting. In both PEXlib 5.2 and OpenGL 1.0, lighting is very simple in concept but extremely subtle in practices.

CHAPTER 6

SHADING

Basic Principles of Shading

Shading is the effect of light on the surface of an object as the surface goes from lighter to darker. In other words shading is the process of applying different shades of color to different pixels as an object is rendered [39]. When one portion of a sphere is darker because the portion faces away from the light, this portion is shaded. If there are several objects in a scene, however, the situation can become more complex, in that each object not only is shaded but might also cast shadows on other objects. All objects in a scene have shading, but not all objects necessarily have cast shadows.

The calculation of shading is fairly straightforward and is provided automatically by all 3D computer graphics systems. However, cast shadows are much more difficult to calculate, and consequently some lower-end system do not allow user to render cast shadows at all, such as PEXlib.

Constant Shading

The simplest shading model for a polygon is constant shading, also called faceted shading, lambent shading or flat shading. This approach applies an lighting model once to determine a single intensity value that is used to shade an entire polygon. Some simplifying assumptions for constant shading are that any flat surface is the same intensity at every point on the surface, and the light source and viewer are at infinity. This is not necessarily true in the physical world, where a flat surface with only one intensity gives objects a faceted look. Constant shading results in very fast renderings. However, constant shading has several limitations. It ignores smoothing information. It also can not receive shadows and can not use bump maps or reflection maps[40].

In order to measure the direction a surface faces, the concept called surface normal is important. A surface normal is a vector perpendicular to the surface at a given point on that surface. A surface normal is usually represented as an arrow coming off the surface. For a flat surface, a single surface normal suffices to indicate the orientation of the entire surface, since all points on a given flat surface face in the same direction.

There are two classic shading methods widely used by 3D computer graphics systems. They are Gouraud shading and

Phong shading. Both are referred to as smooth-shading algorithms. Both of them make use of one critical observation, which is that the surface normals for a smoothly curved surface change gradually over the length of the surface (Figure 23).

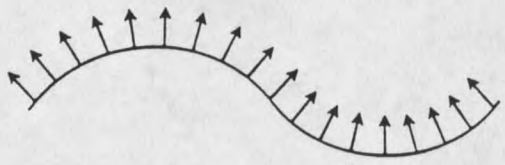


Figure 23. The Normals of a Truly Curved Surface

Gouraud Shading

Gouraud shading [41] is a most common smooth-shading algorithms, and was invented by Henvi Gouraud in 1971. Gouraud shading is also called intensity interpolation shading or color interpolation shading because Gouraud shading eliminates intensity discontinuities by interpolating the intensity for each polygon. Gouraud shading creates a smooth look on a polygonally approximated surface by artificially adjusting the surface normals of the polygons. First, the Gouraud shading algorithm positions a normal at each corner, or vertex, of each polygon. This normal is perpendicular to the surface of the polygon. The normals are easy to get for polygons.

The second step of Gouraud shading is to calculate an average of the normals at the given vertex. The vertex normals, as Gouraud suggested, can be approximated by averaging the surface normals of all polygonal facets sharing each vertex (Figure 24). If an edge is meant to be visible, then two vertex normals can be found, one of each side of edge, by averaging the normals of polygons on each side of the edge separately.

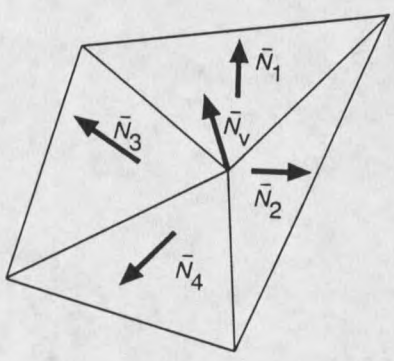


Figure 24. Normalized polygon surface normals may be averaged to obtain vertex normals.

The third step of Gouraud algorithm is to calculate the intensity by using the vertex normals with any light model at each vertex. Once all of these corner intensities have been calculated, the intensities between vertices are calculated by interpolating from one corner intensity to another. Finally, each polygon is shaded by a linear interpolation of the vertex intensities along each edge and then between the edges along each scan line (Figure 25).

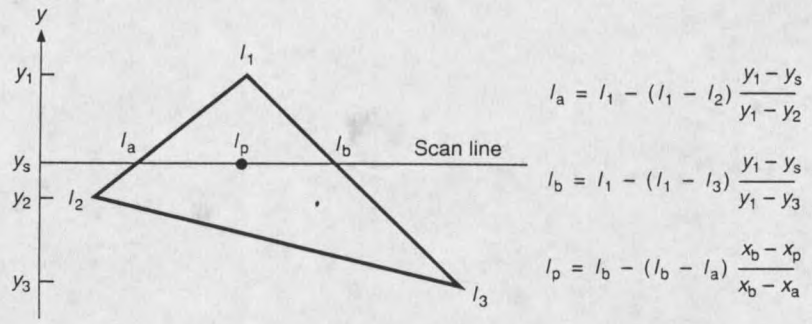


Figure 25. Intensity Interpolation Along Polygon Edges and Scan Lines.

Gouraud shading is very simple, and programs that use the Gouraud-shading algorithm are often very fast. It is useful for non-smoothed objects such as square or rectangular geometries. It also is usually used in applications where speed is important, such as in flight simulators or in interactive application in CAD. Unfortunately, Gouraud shading has some limitations. Most serious, the technique of interpolating one vertex intensity to another sometimes makes the seams between polygons visible. A seam is most apparent when it is near a bright highlight. It means that Gouraud shading will miss the highlight because there is no highlight component at the vertices, even if a polygon contains a highlight (Figure 26). Also, objects can not receive or cast shadows and can not use bump maps or reflection maps with Gouraud shading.

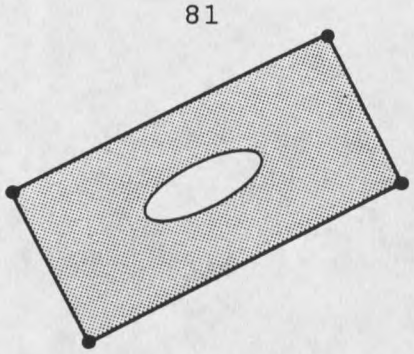


Figure 26. A Polygon that `Contains' A Highlight

Phong Shading

In order to overcome Gouraud shading's limitations, a technique called Phong shading [42] was developed in 1975 by Bui-Tuong Phong. Phong shading averages surface normals at each polygon vertex, just as Gouraud shading does. However, instead of interpolating the intensities between these vertices, Phong shading adds another step which is calculating interpolated surface normals between vertices (Figure 27). From those interpolated surface normals it then calculates the intensities of the surface. This approach yields a more accurate calculation of the intensities, a more accurate rendering of the highlight, and invisible seams along the edges of the polygons. Because Phong shading is expensive, it can take substantially longer rendering a same than Gouraud shading. Phong shading is appropriate for plastic, high-gloss paint, lacquers, glass and other glossy materials that have a harsh glancing highlight.

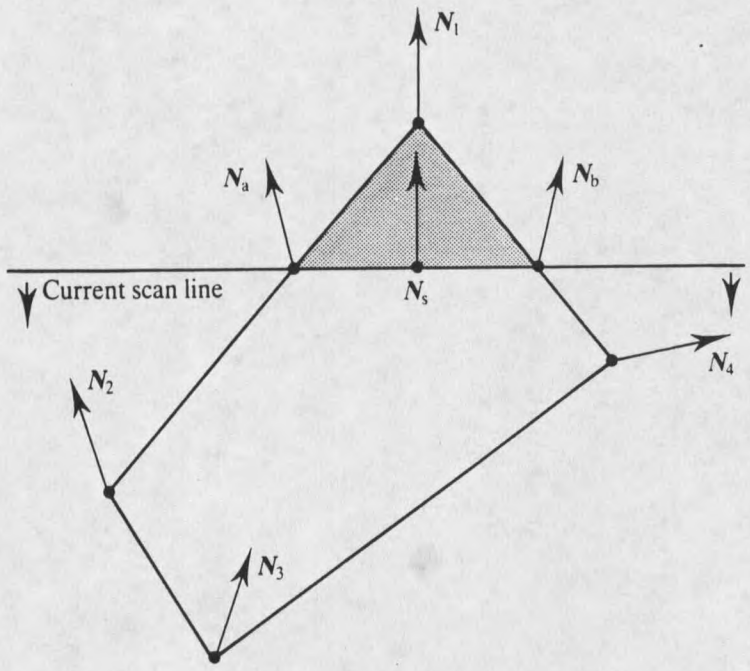


Figure 27. In the Phong method vector interpolation replaces intensity interpolation.

Summary of the Gouraud and Phong Shading

The shading algorithm is not totally separated from the lighting model but often has a certain independence. Gouraud and Phong both have their own characteristics (Figure 28). The motivation of both schemes is efficiency and the `convincing' rendering of polygonal objects. This means that as well as giving an impression of solidity or three-dimensionality to the model, the shared edges in adjacent polygons, that approximate a curved surface, are made invisible. Gouraud and Phong shading are included in GKS3D and PHIGS. Gouraud can be found in direct programmed hardware in many graphics workstations. Gouraud is fast and

simple, and Phong is expensive and produces better results. The two can be successfully merged to produce a useable hybrid. On the other hand, the Gouraud and Phong shading belong to interpolated shading. They both have some common problems which all interpolated-shading models have. They are: polygonal silhouette problems, perspective distortion problems, orientation dependence problems, problems at shared vertices, and unrepresentative vertex normals problems. For more details about these problems, see Foley's Introduction to Computer Graphics on page 496-497.

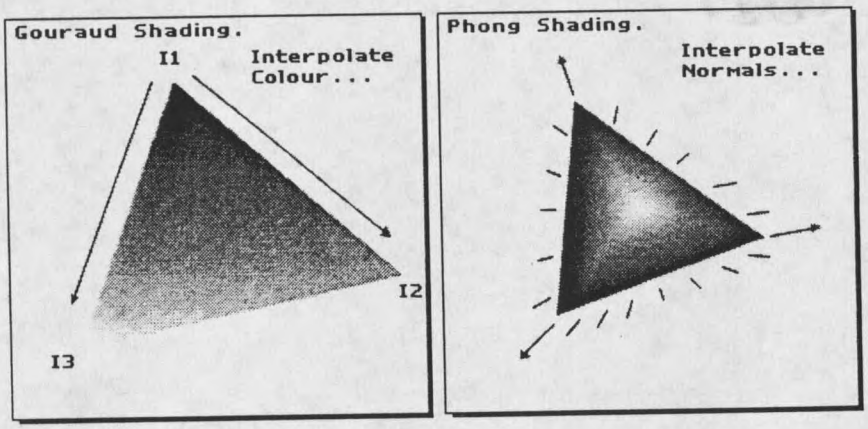


Figure 28. Gouraud and Phong Shading

Shading in PEXlib 5.2

There are four shading methods in PEXlib 5.2. They are: Flat, Gouraud, Dot product and Phong shading, shown in Table 7. They are listed in the order of increasing accuracy at cost of increasing computation. "Dot" shading still belongs to Gouraud shading because it is meant to produce an

improvement in quality over the Gouraud shading method. "Dot" shading also carries a lower cost than reflectance calculations at all interior points. It is seldom available, however, and to many, it is of questionable value [43]. As results, very few PEX implementations support "Dot" shading.

Table 7. Registered Shading Methods in PEXlib 5.2

Common Term	**PEX Mnemonic**
Flat Shading	PEXSurfaceInterpNone
Gouraud Shading	PEXSurfaceInterpColor
	PEXSurfaceInterpDotProudct
Phong Shading	PEXSurfaceInterpNormal

The shading method can be selected with a set of surface interpolation method output commands, which is created with the function PEXSetSurfaceInterMethod in PEXlib 5.2. The most important factor in shading is facet size because small facets make for more accurate scenes. In the back-facing case, PEX 5.2 also provides four shading methods as it does in front-facing case, and uses the PEXSetSurfacebfInterMethod function.

Summary of the Shading System in PEXlib 5.2

In short, PEXlib 5.2 provides a nice shading system which is easy to use. On the other hand, there are some limitations of the shading system in PEXlib 5.2 as we mentioned before. For example, it does not compute shadows or follow light as it bounces from one object to another.

Shading in OpenGL 1.0

Since the design strategy is simple and fast to render a scene in OpenGL, OpenGL 1.0 supports only two types of shading: flat shading and Gouraud shading, and specifies the desired shading technique with the function glShadeModel(). However, it doesn't support Phong shading because SGI thinks that Phong shading is considerably more expensive to implement, in terms of either silicon area or computation time.

With flat shading, the intensity of one vertex of a primitive is duplicated across all of the primitive's vertices. For a line segment, the intensity of the line is the current intensity when the second (ending) vertex is specified. For a polygon, the intensity used is the one that is in effect when a particular vertex is specified, as shown in Table 8. The table counts vertices and polygons starting

from 1. OpenGL 1.0 follows these rules consistently, but the best way to avoid uncertainty about how flat shading primitive will be drawn is to specify only one intensity for the primitive [44].

Table 8. How OpenGL 1.0 Selects a Color for the ith Flat-Shaded Polygon

Type of Polygon	Vertex Used to Select the Color for the ith Polygon
single polygon	1
triangle strip	i+2
triangle fan	i+2
independent triangle	3i
quad strip	2i+2
independent quad	4i

With Gouraud shading, the intensity at each vertex is treated individually. For a line primitive, the intensities along the line segment are interpolated between the vertex intensities. For a polygon primitive, the intensities for the interior of the polygon are interpolated between the vertex intensities. Also, in RGBA mode, adjacent pixels with slightly different values look similar, so the intensity changes across a polygon appear gradual. However, in color index mode, adjacent pixels may reference different locations in the color index table, which may not have similar colors at all. Adjacent color index entries may

contain wildly different colors, so a Gouraud-shaded polygon in color index mode can look psychedelic, To avoid this problem, it is helpful to create a color ramp of smoothly changing colors among a contiguous set of indices in the color map.

Summary of the Shading System in OpenGL 1.0

The shading system is simple and fast for OpenGL 1.0. Unfortunately, it doesn't support Phong shading and shadows. It still can provide acceptable scenes for most applications that are rapidly rendered.

Analysis

As discussed already, PEXlib 5.2 and OpenGL 1.0 both focus on Gouraud shading for their primely shading method. However, PEXlib 5.2 provides a wider range of shading method than OpenGL 1.0. At this point, there is a problem in PEXlib 5.2. With PEX 5.2, the problem is that how many shading methods can be used is really depending on the PEX 5.2 implementation. In other words, the problem is that each PEX system can define its own PEX implementation. With OpenGL, every user has the same implementation as the standard one. As a result, from the user's view, PEXlib 5.2 and OpenGL 1.0 provide almost the same features for their shading systems.

CHAPTER 7

VIEWING

Basic Principles of Viewing

Before this chapter, the main focus is on how to draw the geometry with colors, lighting and shading effects. In this chapter, the focus will change to how to display the scene that is already drawn. In other words, the focus will care about what and how much the viewer can see. There are some basic principles related to the viewing system, such as coordinate systems, screen space, projections and the camera analogy, etc. The conceptual model of the 3D viewing process is shown in Figure 29 [45].

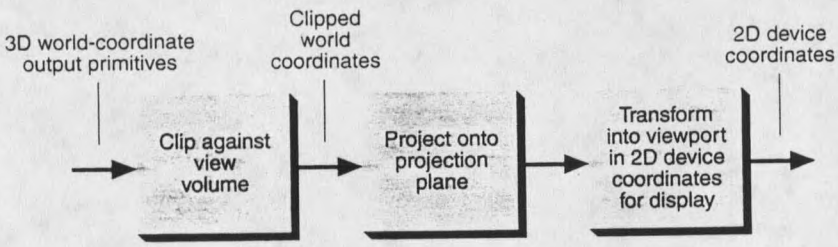


Figure 29. Conceptual Model of The 3D Viewing System

Virtual Camera

A virtual camera is often used as a conceptual aid in computer graphics. The camera can be positioned anywhere in world space and pointed in any direction. The analogy of the film plane in computer graphics is the view plane which is the plane on which the scene is projected. Additionally, the camera is allowed to be rotated about the view direction. The camera, of course, is really just a computer program that produces an image on a display screen, and the object is a 3D dataset comprising a collection of points, lines and surfaces. There is a coordinate system for the virtual camera. It is called (U, V, N) system. The (U, V, N) system is from the PHIGS viewing system. A camera with (U, V, N) coordinate system is shown in Figure 30. The (U, V, N) coordinate also can be called the eye coordinate system. It is specified by three vectors and a position.

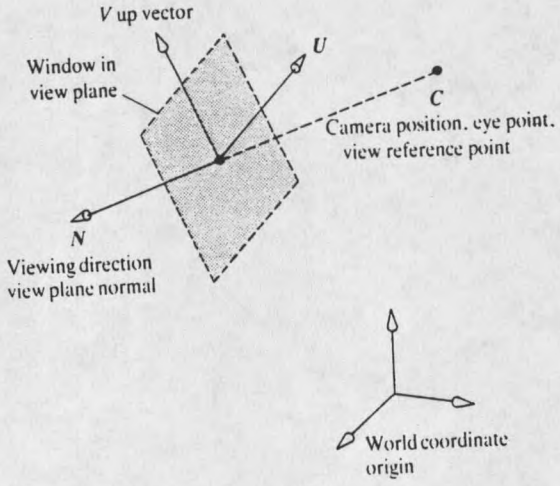


Figure 30. A Camera with (U,V,N) System

In Figure 30, C is a virtual camera position. This point is also the center of projection. N is a viewing direction vector, and a vector normal to the view plane. V is an up vector that orients the camera about the view direction and establishes with N the orientation of the view plane window within the view plane. U is a vector which denotes the direction of increasing X. In comparison to the world coordinate system, U is same as X axis, N is the positive Z-axis and V is the same as the Y axis. The virtual camera is a useful concept, but it is not enough to produce an image.

Projection

After the virtual camera is specified, the next concept is the projections in the 3D viewing system. The projection of a 3D object is defined by straight projection rays, called projectors, emanating from a center of projection, passing through each point of the object, and intersecting a projection plane to form the projection[46]. The class of projections is the planar geometric projections, because the object needs to be converted from 3D to 2D. The projection is onto a plane rather than onto a curved surface, and uses straight rather than curved projectors.

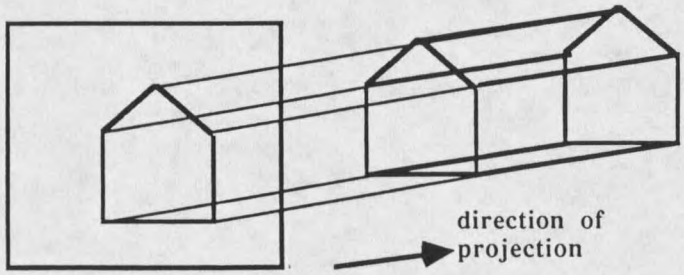


Figure 31. Principle of Parallel Projection

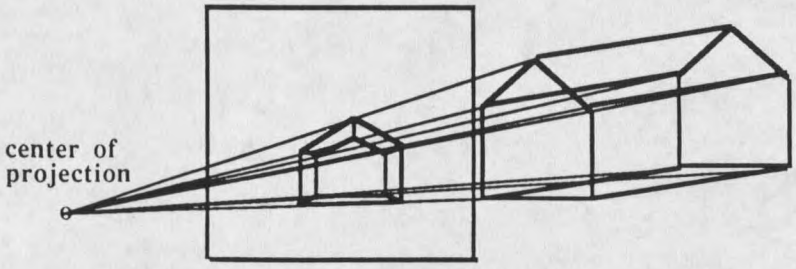


Figure 32. Principle of Perspective Projection

There are two kinds of planar projections — parallel projections (Figure 31) and perspective projections (Figure 32). In both cases, the object projection is obtained by projectors from the points of the object and by looking for the intersections of these projectors with the view plane. In the case of a perspective, all projectors come from a single point that is the center of projection. This point also is a finite distance away from the view plane. For a parallel projection, all projectors are parallel to the direction of projection. The center of projection is

infinitely far away from view plane. The center of projection and direction of projection (DOP) are defined by a projection reference point (PRP) and an indicator of the projection type. If the projection type is perspective, then PRP is the center of projection. If the projection type is parallel, then the DOP is from the PRP to CW (Center of Window). There are two cases for parallel projection. If the direction of projection is perpendicular to the view plane, the projection is orthographic; otherwise it is oblique. For more details about projection, see Foley's Introduction To Computer Graphics on page 195-200.

The view plane is a plane which requires projections of the objects onto it from 3D to 2D. It is defined by a point on the plane called the view reference point (VRP) and a normal to the plane called the view plane normal (VPN), as shown in Figure 33. The view plane may be anywhere with respect to the world objects to be projected. It may be in front of, cut through, or be behind the objects.

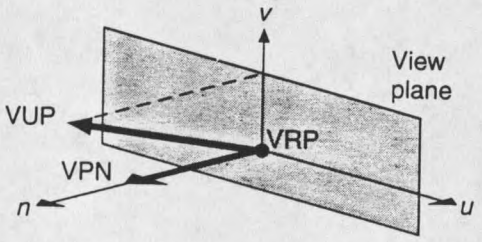


Figure 33. The view plane is defined by VPN and VRP

When the view plane is defined, a window on the view plane is needed. The window's role is similar to that of a 2D window. Its contents are mapped into the viewport, and any part of the 3D world that projects onto the view plane outside of window is not displayed. To define a window on the view plane, (Figure 34)[47], there are things to be specified: the minimum and the maximum window coordinates and two orthogonal axises in the view plane along which to measure these coordinates.

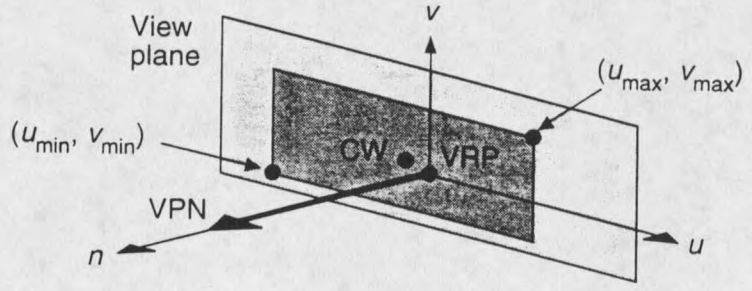


Figure 34. The View Plane

Coordinate System

So far the coordinate system has not been discussed yet, but it is a really important part for the 3D viewing system. There are several coordinate systems, such as the local coordinate system (LCS), the normalized-projection coordinate system (NPC) or 3D screen coordinate system, world coordinate system and the view-reference coordinate system (VRC). Although there are several coordinate systems, the commonly used are the world coordinate system, view-

reference coordinate system and normalized-projection coordinate system. Figure 35 shows how they are related to each other in the 3D viewing system.

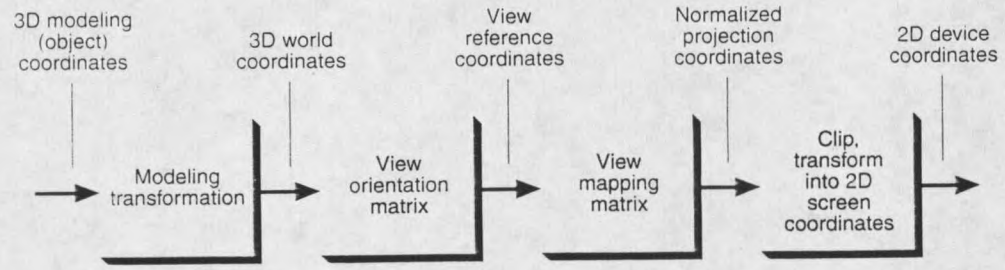


Figure 35. Coordinate System and How they relate to one another.

The world coordinate system is the most commonly used system, and is easy to understand. The view-reference coordinate (VRC) system is not as easy as world coordinate system (Figure 36). It is used to define a view volume, and also define the (U, V, N) system discussed before.

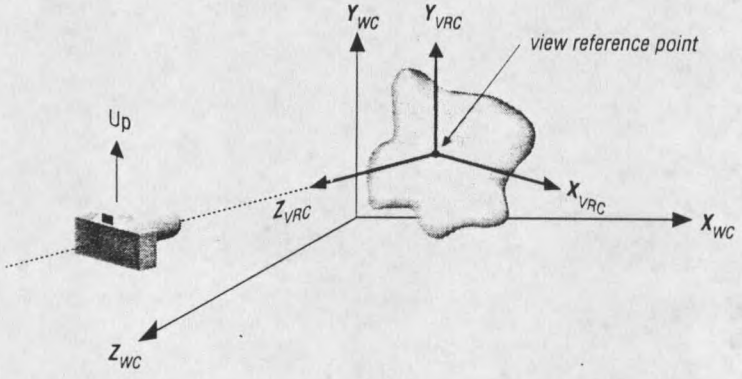


Figure 36. The View Reference Coordinate System

The origin of VRC system is the VRP. One axis of the VRC is VPN. A second axis of the VRC is VUP (View-up Vector). The third axis of the VRC is defined so that the VPN, VUP and the third axis from a right-handed coordinate system (Figure 37). In the VRC, everything is relative to the viewer. The normalized-projection coordinate (NPC) system is a coordinate system which is used to normalize the view volume in some prespecified range.

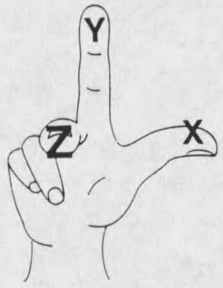


Figure 37. Right-handed Coordinate System

Steps for Defining a View

After several important concepts of 3D viewing system have been presented, to define a view becomes simple. Table 9 shows the steps which are used to define a view in a 3D system. Although the viewing system is more complex than lighting and shading in 3D graphics system, it is a necessary part for a 3D graphics API which is used to simulate our real world within a screen. Since the viewing system is a big part for 3D graphics system, there are a lot of concepts involved that can not be covered in this thesis.

For more details about viewing, see Foley's Introduction To Computer Graphics and Watt's Advanced Animation and Rendering Techniques — Theory and Practice.

Table 9. Steps Used to Define a View in a 3D System

Step	Function	Coordinate System
1	Establishing the VRC system	WC
2	Establishing the location for virtual camera	VRC
3	Selecting the type of projection to be used	VRC
4	Specifying the clipping that should occur	VRC
5	Defining the way to place the view in a normalized viewing plane and window	NPC

Viewing in PEXlib 5.2

In PEXlib 5.2, viewing is determined by the view volume and the viewport. The view volume is defined by two matrices: the view orientation and the view mapping matrix. Views are maintained in the view table which is an attribute of the renderer resource. The view table entry consists of the viewing matrices, the viewport bounds in NPC and a set of flags to tell which of the viewport boundaries to clip against. Figure 38 shows a view table which is just another instance of a PEX lookup table. It is nice for multiple views because multiple views can be constructed and saved in

the view table, and applications can specify different views of the scene by just changing the view table entry used.

View Table				
Index	View Definition			
0	[Orientation Matrix]	[Mapping Matrix]	Clip limits (min.x, max.x) (min.y, max.y) (min.z, max.z)	Clip flags PEXClipFront | PEXClipBack | PEXClipXY
1	[Orientation Matrix]	[Mapping Matrix]	Clip limits (min.x, max.x) (min.y, max.y) (min.z, max.z)	Clip flags PEXClipFront | PEXClipBack | PEXClipXY
N	[Orientation Matrix]	[Mapping Matrix]	Clip limits (min.x, max.x) (min.y, max.y) (min.z, max.z)	Clip flags PEXClipFront | PEXClipBack | PEXClipXY

Figure 38. View Table

The VRC coordinate system defines what PEXlib 5.2 calls the view orientation. View orientation can be set with the PEXlib 5.2 utility function PEXViewOrientationMatrix. When the view orientation is defined, there are a lot of things that are still not clear, such as where is the virtual camera, what type of projection will be used and how much the model can be seen. All of these are specified in the view mapping whose job is to convert all or a portion of scene to a flat image. PEXlib 5.2 provides two types of projection: parallel and perspective.

After the view orientation and view mapping defined, the view volume has five factors that determine the size and shape of the view volume. They are: front plane, back plane, view plane, view window and projection reference point. These factors determine both view volume and the projected image itself by taking them together. Objects outside the view volume are normally not displayed. Also, it is changeable.

The view mapping need not map the entire view volume to the entire NPC unit cube. The view volume can be mapped to any portion of NPC that is called a projection viewport. A window can be divided into regions and display a different view in each region by defining each view so that it maps to a different projection viewport. It is often useful to see a scene with different views in a same window.

PEX 5.2 applies the view orientation and view mapping transforms to the entire scene, not just the primitives within the view volume. Primitives outside the view volume are also transformed, but end up in NPC positions outside the projection viewport. If the projection viewport is less than all of the NPC, then these primitives outside the view volume are displayed, unless user wants to clip them. Most programmers don't want primitives outside the view volume to be visible. View clipping is used to control this. View clipping is governed with clip limits and clip flags. Clip limits are specified in NPC and indicate the X, Y, and Z

clipping boundaries. Clip flags determine which clip limits are active. There are three clip flags: front limit flag, back limit flag and side limits flag.

Although viewing is determined by the view volume and the viewport, in general, there are six steps involved for defining a view in PEXlib 5.2 [48]. They are:

(1) establishing the VRC system
(2) establishing the virtual camera position
(3) specifying the projection to be used
(4) specifying the clipping that should occur
(5) specifying how to place the view in a NPC cube
(6) saving the information about the view in a view table

The ordering of these steps are shown above. After these steps are performed, the scene can be drawn on the screen.

Summary of the Viewing System in PEXlib 5.2

Since there are a lot of things involved which viewing in PEXlib 5.2, some of them can not be covered here. But the basic ideas are very closely to the basic principles of viewing. For more details, see the PEXlib Programming Manual by Tom Gaskins on page 547-630.

Viewing in OpenGL 1.0

The OpenGL 1.0 viewing system is almost the same as the PEXlib 5.2 viewing system. The viewing is also determined by the view volume and the viewport. The virtual camera and NPC are used too. As usual, the OpenGL 1.0 provides two types of projection: perspective and orthographic. However, there is a little bit difference with PEXlib 5.2 because PEXlib 5.2 provides parallel projection which includes orthographic and oblique cases. The main reason that OpenGL 1.0 provides orthographic is that it maps objects directly onto the screen without affecting their relative size, and is useful for architectural and CAD applications where the final image needs to reflect the measurements of objects rather than how they might look.

The central idea in the OpenGL 1.0 viewing system is transformations. Transformations are represented by matrix multiplication, including modeling, viewing and projection operations. Such operations include rotation, translation, scaling, reflecting and two types of projection. Generally, a combination of several transformations will be used to draw a scene. Depending on transformations, there are four steps involved for defining a view in OpenGL 1.0 [49]. They are:

(1) Viewing transformation — setup the virtual camera

and VRC system, positioning the viewing volume in the world

(2) Modeling transformation — positioning the models in the world (arranging the scene to be viewed into the desired composition)

(3) Projection transformation — determining the shape of the viewing volume

(4) Viewport transformation — determining how large the final scene can be seen

The order of these steps are not necessarily related to the relevant mathematical operations which are performed on an object's vertices. Their order really depends on the desired transformations which are specified by users. The viewing transformations must precede the modeling transformations, but the projection and viewport transformations can be specified at any point before drawing occurs.

The clipping operation in OpenGL 1.0 is not as same as that in PEXlib 5.2. In OpenGL 1.0, when a viewing volume is defined, objects outside this volume are clipped so that they are not drawn in the final scene. OpenGL 1.0 reconstructs the edges of polygons that get clipped. There are six clipping planes of the view volume (left, right, bottom, top, near, and far). The six clipping planes used are those that define the sides and ends of the view volume. Also, additional clipping planes can be specified anywhere.

The clipping operation happens after the view volume defined in OpenGL 1.0. However, the clipping operation happens when the scene is in NPC. The OpenGL's clipping operation is faster than PEXlib's.

Analysis

Both the PEXlib 5.2 viewing system and the OpenGL 1.0 viewing system follow the basic concepts of viewing very closely. For example, they both use the concepts of virtual camera, view volume, viewport and projection. Although OpenGL's viewing system is faster than PEXlib's, the PEXlib 5.2 viewing system has some advantages over OpenGL 1.0 viewing system, such as multiple view operation and different views for the same scene in a screen. The multiple view operation and different views for a same scene in a screen can also be presented by OpenGL 1.0, but it is more complicated to do it in OpenGL 1.0 than in PEXlib 5.2. By contrast, PEXlib 5.2 viewing system is a system that cares about performance and integrates theory with practice. The OpenGL 1.0 viewing system is a system that focuses more on applications and effects.

CHAPTER 8

MISCELLANEOUS FEATURES AND PORTABILITY

Miscellaneous Features for PEXlib 5.2 and OpenGL 1.0

The main features for PEXlib 5.2 and OpenGL 1.0 have been discussed in the earlier chapters. Although the main features define a system's properties, some miscellaneous features can also add a lot of credits and provide a lot of convenience. PEX 5.2 does support features not available in OpenGL 1.0. PEX 5.2 has extensive text support for stroke font which are fully transformable in 3D. B-spline surface and curves are supported directly by PEX 5.2 while OpenGL 1.0 supports NURBS (Nonuniform Rational B-splines) functionality via routines which are part of the GLU (Graphics Library Utility) library. PEX 5.2 can support cell arrays but the functionality is seldom implemented. Markers and quadrilateral meshes are supported by PEX 5.2 as a rendering primitive; neither are supported as primitives by OpenGL 1.0. PEX 5.2 support self-intersecting contours and polygon lists with shared geometry, while OpenGL 1.0 does not. OpenGL 1.0 also provides some atmospheric effects features (e.g., Fog and Haze).

Table 10. The Miscellaneous Features for PEXlib and OpenGL

Features	PEXlib 5.2	OpenGL 1.0	Remarks
Immediate Rendering	Yes	Yes	OpenGL provides faster rendering than PEXlib.
Database Rendering	Yes	No	
Mixed Rendering	Yes	N/A	
Draw line/polygon, etc.	Yes	Yes	OpenGL provides the easiest way to draw geometric objects.
Marker	Yes	No	
Text	Yes	Yes	PEXlib does better job than OpenGL in this area.
Complex Polygons	Yes	Yes	PEXlib does better job than OpenGL because PEX supports self-intersecting contours and polygon list with shared geometry.
Alpha Blending	Yes*	Yes*	
Double Buffering	Yes*	Yes*	
Image Operation	No	Yes	
Texture Mapping	Yes	Yes	
Picking	Yes	Yes	
Atmospheric Effects	No	Yes	
Z-Buffer	Yes	Yes	
Antialiasing	Yes*	Yes*	
Motion Blur	No	Yes	
Depth Cueing	Yes*	Yes*	
Communication	Yes	Yes	PEXlib does better job than OpenGL.
Language Binding	Yes	Yes	PEXlib binds with C, OpenGL binds with C, C++, FORTRAN, Ada and Java.
X-UNIX	Yes	Yes	PEXlib is nice to use with X.
window/NT 3.5	No	Yes	
OS/2	No	Yes	

* — The quality of this feature is dependent on PEX/OpenGL vendor.

Table 10 shows some of the miscellaneous features which are frequently used by applications. For more details about these features, see the PEXlib Specification and C Language

Binding — Version 5.2, and the OpenGL Graphics System: A Specification (Version 1.0).

Portability

In all of the software development cycles, portability is a primary concerned. Portability is described as the ability to carry or move something. For graphics systems or software, portability means the ease with which graphic system or software can be transferred from one computer system or environment to another (IEEE Dictionary 1984). And most portability problems are not pure software issues. portability involves hardware (byte order, device I/O) and software (utility libraries, operating systems and runtime libraries).

While PEX was designed to be vendor-independent and portable, the subsetting allowed by the PEX standard allows implementations of greatly varying functionality to claim to be "standard" PEX implementations. The fact that PEX explicitly allows multiple subsets perhaps indicate the PEX standard may be too large to implement fully and completely in a timely fashion.

OpenGL does not allow any subsetting of rendering functionality and therefore can expect much greater application portability. The need for interpretability

testing for OpenGL is greatly reduced because OpenGL demands more consistent implementations.

Neither OpenGL 1.0 nor PEXlib 5.2 is pixel exact. This means neither specification is completely explicit about what pixels must be modified by each rendering operation (the core X protocol is largely pixel exact). Pixel exactness is not a totally desirable feature for 3D since much 3D graphics is done with floating point where numerical errors make exactness nearly impossible. But the OpenGL 1.0 specification is much more rigorous than PEXlib 5.2 about what is considered conformist behavior. Also, the OpenGL release kit includes a suite of conformance tests to verify rendering accuracy. No comprehensive test suites are yet available to validate PEXlib implementations.

PEXlib 5.2 as it is defined today is very closely coupled to the X Window System. In the Unix world, the portability for PEXlib 5.2 is a little bit better than OpenGL 1.0. However, there are other commercially-significant window systems. Microsoft Windows is an example. In order to accommodate these systems, OpenGL is designed as a window independent system. This capability is being exploited to port OpenGL 1.0 to Microsoft Windows/NT and other windows systems. In a word, OpenGL 1.0 has better portability than PEXlib 5.2 because of the architecture of the OpenGL 1.0. Also, this is the main reason why OpenGL 1.0

is widely available. Hopefully, the portability is a goal for PEXlib 5.2 which needs to improve in later releases.

CHAPTER 9

CONCLUSIONS AND RECOMMENDATIONS

Conclusions

PEXlib 5.2 and OpenGL 1.0 have been discussed by this thesis based on the basic graphic theories. Since PEXlib 5.2 and OpenGL 1.0 both belong to low-level 3D APIs, their purposes are to draw objects. High-level tasks such as object editing and file I/O are left up to the applications. There are more than 50 APIs in the 3D graphics markets, such as Quick Draw 3D from Apples, Direct3D from Microsoft and HOOPS(Hierarchical Object-Oriented Programming System) from Auto Desk [50]. It is becoming more and more important to choose the right 3D graphics API.

As Dataquest (San Jose) reported in 1994, about one-third of CAD/CAM/CAE users planned purchase in the next two years. About one-third said they planned to purchase integrated systems that provided solids-modeling, surfacing and 2D, 3D wire-frame capabilities. Another third said they planned to buy products based on a specific 3D technology, such as PEXlib and OpenGL. The remaining 39 percent said they would be purchasing only 2D based products [51].

With the development of computer technology going rapidly, the 3D APIs were requested from both users and vendors sides. But PEXlib 5.2 and OpenGL 1.0 still have some problems. For example, one vendor's PEX implementation will not be compatible with an other's, preventing PEX-based applications from running on multiple platforms [52]. Rendering speed and data duplication also are problems for PEXlib 5.2. On the other hand, OpenGL 1.0 is less focused on client/server communications than on graphic effects [53]. Although Microsoft's official line is that the OpenGL 1.0 is 3D standard for Windows/NT 3.5, there are still some questions [54].I.e., Microsoft says yes but some insiders have doubts. OpenGL 1.0 can be run on PC environment but the performance is questionable.

Although PEXlib 5.2 and OpenGL 1.0 still have some problems, they are two major 3D APIs in the 3D graphics markets. They both bring out a lot of nice features and conveniences for users to design their own scene. Also, they both have their application areas. For example, PEXlib's applications are found in CAD/CAM, government, European markets and 3D graphics research[55], while OpenGL is usually used in CAM/CAE, film animation and scientific visualization, such as microbiology research.

Recommendations

Because PEXlib 5.2 and OpenGL 1.0 are two big and complex 3D API packages, it is hard to be familiar with everything in both packages. As mentioned before, more practice can get more benefits from both systems.

REFERENCES CITED

1. Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L., "Introduction to Computer Graphics", PP. 1, Addison-Wesley Publishing Company, 1994.

2. Segal, M., Akeley, K., "The OpenGL Graphics System : A Specification (Version 1.0)" SGI, 1995.

3. International Standards Organization, "International Standard Information Processing Systems — Computer Graphics — Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description", Technical Report ISO Document Number 9905:1988(E), American National Standards Institute, New York, 1988.

4. Shamansky, H., "OpenGL — The Integration of Windowing and 3D Graphics", PP. 1, hts@hertz.eng.ohio-state.edu.

5. Thomas, S.W., "PEX-A 3D Extension to X Windows", *USENIX-Winter'89*, PP.139-149, Dec., 1988.

6. Hardenbergh, J.Y.,"Building Applications with PEXlib", PP. ix, P T R Prentice Hall, New Jersey, 1994.

7. Harpenbergh, J. Y., "Building Applications with PEXlib", PP. x, P T R Prentice-Hall, New Jersey, 1994.

8. Kaufman, J. and Schultz, K., "PEXlib Specification and C Language Binding", Version 5.2 — Draft Consortium Standard — Rev. 3, Jan. 10, 1996.

9. Chatterjee, S., "OpenGL Programs a New Horizon for Sun" *Sun World* PP. 15-20, Jan. 1994.

10. Prosise, J., "Advanced 3-D Graphics for WindowsNT 3.5 — Introducing the OpenGL Interface", Part 1, *MICROSOFT Systems Journal*, PP. 15-27, Oct. 1994.

11. Segal, M. and Akeley, K., "The OpenGL Graphics Interface", SGI.

12. Foust, J., "3D Steps Forward", *Byte*, PP. 123-130, July 1995.

13. Glazier, B., "The "Best Principle" - Why OpenGL is Emerging as The 3D Graphics Standard", *Computer Graphics World*, PP. 115-116, April 1992.

14. Segal, M. and Akeley, K., "The Design of the OpenGL Graphics Interface", PP. 5, SGI, 1994.

15. "OpenGL 3D Graphics", *Microsoft* , 1995.

16. Gaskins, T., "PEXlib Programming Manual", O'Reilly & Associates, Inc., PP. 19, Dec. 1992.

17. Rost, R.J., "PEX: A Network-Transparent 3D Graphics System", DEC, *IEEE Computer Graphics & Applications*, July 1989.

18. Voorhies, D., "Damn the Torpedoes", *Byte*, PP. 224, Nov. 1993.

19. Williams, T., "Industry Group Pushing 3D Graphics Standard", *Computer Design*, PP. 50, July 1994.

20. Neider, J., Davis, T., and Woo, M., "OpenGL Programming Guide", PP. 121, June 1995.

21. Segal, M. and Akeley, K., "The Design of the OpenGL Graphics Interface", PP. 7, SGI, 1994.

22. Shamansky, H., "OpenGL - The Integration of Windowing and 3D Graphics", PP. 4, hts@hertz.eng.ohio-state.edu, Dec. 1995.

23. Kilgard, M.J., Hui, S., Leinwand, A.A., and Spalding D., *The X Resource Issue*, **9**, PP. 73-88, Jan. 1994.

24. Foley, J.D. and Grimes J., "Using Color in Computer Graphics", *IEEE Computer Graphics & Applications*, PP. 25-27, Sept. 1988.

25. Durrett, H.J., "Color and the Computer", PP. 27, Academic Press, 1987.

26. Harpenbergh, J. Y., "Building Applications with PEXlib", PP. 171, P T R Prentice-Hall, New Jersey, 1994.

27. Gaskins, T., "PEXlib Programming Manual", PP. 185, O'Reilly & Associates, Inc., 1992.

28. Gaskins, T., "PEXlib Programming Manual", PP. 163, O'Reilly & Associates, Inc., 1992.

29. Neider, J., Davis, T., and Woo, M., "OpenGL Programming Guide", PP. 140, 1995.

30. Neider, J., Davis, T., and Woo, M., "OpenGL Programming Guide", PP. 145, 1995.

31. O'Rourke, M., "Principles of Three-Dimensional Computer Animation", PP. 83, W. W. Norton & Company, 1995.

32. O'Rourke, M., "Principles of Three-Dimensional Computer Animation", PP. 79, W. W. Norton & Company, 1995.

33. Harpenbergh, J. Y., "Building Applications with PEXlib", PP. 172-173, P T R Prentice-Hall, New Jersey, 1994.

34. Schultz, K., Kaufman, J., "PEXlib Specification and C Language Binding-Version 5.2", PP. 834, MIT, Jan. 1996.

35. Harpenbergh, J. Y., "Building Applications with PEXlib", PP. 179, P T R Prentice-Hall, New Jersey, 1994.

36. Segal, M., Akeley, K., "The OpenGL Graphics System : A Specification (Version 1.0)", 2.12.2, SGI, 1995.

37. Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L., "Introduction to Computer Graphics", PP. 485, Addison-Wesley Publishing Company, 1994.

38. Watt, A. and Watt., M., "Advanced Animation and Rendering Techniques — Theory and Practice", PP. 42, Addison-Wesley Publishing Company, 1993.

39. Harpenbergh, J. Y., "Building Applications with PEXlib", PP. 183, P T R Prentice-Hall, New Jersey, 1994.

40. O'Rourke, M., "Principles of Three-Dimensional Computer Animation Modeling, Rendering and Animating with 3D Computer Graphics", PP. 83, W. W. Norton & Company, 1995.

41. Gouraud, H., "Continuous Shading of Curved Surfaces", *IEEE Trans. on Computers*, **C-20(6)**, PP. 623-629, June 1971.

42. Phong, B.T., "Illumination for Computer Generated Pictures", *CACM*, **18(6)**, PP. 311-317, June 1975.

43. Gaskins, T., "PEXlib Programming Manual", PP. 763, O'Reilly & Associates, Inc., 1992.

44. Neider, J., Davis, T., and Woo, M., "OpenGL Programming Guide", PP. 154-155, Addison-Wesley Publishing Company, 1995.

45. Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L., "Introduction to Computer Graphics", PP. 195, Addison-Wesley Publishing Company, 1994.

46. Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L., "Introduction to Computer Graphics", PP. 196, Addison-Wesley Publishing Company, 1994.

47. Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L., "Introduction to Computer Graphics", PP. 202, Addison-Wesley Publishing Company, 1994.

48. Babcock, R.S. and Starkey, J.D., "PEXlib by Example: A Gentle Introduction", PP. 54, Montana State University, Fall 1993.

49. Neider, J., Davis, T., and Woo, M., "OpenGL Programming Guide", PP. 66-69, Addison-Wesley Publishing Company, 1995.

50. Thompson T., "Must-See 3-D Engines", *Byte*, PP. 137-144, June 1996.

51. Bruno, L., "Graphics Users Debate Three Hot Topics", *Open Systems Today*, Dec. 1994.

52. Bruno, L., "Sun Continues To Resist OpenGL Tide", *Open Systems Today*, Nov. 1994.

53. Simth, B., "Battle of 3D Space Libraries", *Byte*, PP. 42, May 1994.

54. Baran, N., "Will OpenGL Be 3D Standard for Windows/NT?", *Byte*, PP. 144-146, Nov. 1993.

55. Akin, A., "Analysis of PEX 5.1 and OpenGL 1.0", SGI, Aug. 1992.

MONTANA STATE UNIVERSITY LIBRARIES
3 1762 10240427 2