IRIS Performer™
Programmer's Guide

Document Number 007-1680-030

CONTRIBUTORS

Edited by Steven Hiatt
Illustrated by Dany Galgani
Production by Derrald Vogt
Engineering contributions by Sharon Clay, Brad Grantham, Don Hatch, Jim Helman, Michael Jones, Martin McDonald, John Rohlf, Allan Schaffer, Chris Tanner, and Jenny Zhao

IRIS Performer™ Programmer's Guide
Document Number 007-1680-030

Contents

Examples

Figures

Tables

About This Guide

Welcome to the IRIS Performer™ application development environment. IRIS Performer provides a programming interface (with ANSI C and C++ bindings) for creating real-time graphics applications and offers high-performance rendering in an easy-to-use 3D graphics toolkit. IRIS Performer interfaces to both the IRIS Graphics Library™ (also known as IRIS GL™) and the OpenGL® graphics library; these libraries combine with the IRIX™ operating system to form the foundation of a powerful suite of tools and features for creating real-time 3D graphics applications on Silicon Graphics systems.

Why Use IRIS Performer?

Use IRIS Performer for building visual simulation applications and virtual reality environments, for rapid rendering in on-air broadcast and virtual set applications, for assembly viewing in large simulation-based design tasks, or to maximize the graphics performance of any application. Applications that require real-time visuals, free-running or fixed-frame-rate display, or high-performance rendering will benefit from using IRIS Performer.

IRIS Performer drastically reduces the work required to tune your application's performance. General optimizations include the use of highly-tuned routines for all performance critical operations and the reorganization of graphics data and operations for faster rendering. IRIS Performer also handles Silicon Graphics architecture-specific tuning issues for you by selecting the best rendering and multiprocessing modes at run time based on the system configuration.

IRIS Performer is an integral part of the RealityEngine™ and Impact™ visual simulation systems and provides the interface to advanced features available exclusively with RealityEngine graphics. IRIS Performer teamed with RealityEngine or Impact provides a sophisticated image generation

system in a powerful, flexible, and extensible software environment. IRIS Performer is also tuned to operate at peak efficiency on each graphics platform produced by Silicon Graphics; you don't need the hardware sophistication of RealityEngine graphics to benefit from IRIS Performer.

What You Should Know Before Reading This Guide

To use IRIS Performer, you should be comfortable programming in ANSI C or C++. You should have a fairly good grasp of graphics programming concepts (terms such as "texture map" and "homogeneous coordinate" aren't explained in this guide) and it will help if you're familiar with at least one of the graphics libraries. If you're a newcomer to these topics, see the references listed under "Bibliography" at the end of this introduction and examine the glossary for definitions of terms or usage unique to IRIS Performer.

On the other hand, though you need to know a little about graphics, you don't have to be a seasoned C (or C++) programmer, a graphics hardware guru, or a graphics-library virtuoso to use IRIS Performer. IRIS Performer puts the engineering expertise behind Silicon Graphics hardware and software at your fingertips, so you can minimize your application development time while maximizing the application's performance and visual impact.

How to Use This Guide

The best way to get started is to turn to Chapter 1, "Getting Acquainted With IRIS Performer," which takes you on a tour of some demo programs. These programs let you see for yourself what IRIS Performer does. Even if you aren't developing a visual simulation application, you might want to look at the demos to see high-performance rendering in action. At the end of Chapter 1 you'll find suggestions pointing to possible next steps; alternatively, you can browse through the summary below to find a topic of interest.

What This Guide Contains

This guide is divided into fourteen chapters and an appendix:

- Chapter 1, “Getting Acquainted With IRIS Performer,” takes you on a tour of some demonstration databases using sample IRIS Performer applications.
- Chapter 2, “IRIS Performer Basics,”provides a brief overview of IRIS Performer—its features, applications, and component libraries—as well as a brief survey of general visual simulation techniques.
- Chapter 3, “Building a Visual Simulation Application,” outlines the structure of a visual simulation application.
- Chapter 4, “Setting Up the Display Environment,” describes how to set up rendering pipelines, windows, and channels (cameras).
- Chapter 5, “Nodes and Node Types,” describes the data structures used in IRIS Performer’s memory-based scene-definition databases.
- Chapter 6, “Database Traversal,” explains how to manipulate and examine a scene graph.
- Chapter 7, “Frame and Load Control,” explains how to control frame rate, synchronization, and dynamic load management. This chapter also discusses the load management techniques of multiprocessing and level-of-detail.
- Chapter 8, “Creating Visual Effects,” describes special features used in visual environments, including spotlights, shadows, realistic earth and sky models, and morphing.
- Chapter 9, “Importing Databases,” describes database formats and sample conversion utilities.
- Chapter 10, “libpr Basics,” discusses the foundation on which IRIS Performer is based and fundamental concepts for working with its basic building blocks, the functions of the low-level library *libpr* that gives IRIS Performer its speed.
- Chapter 11, “Math Routines,” details the comprehensive math support provided as part of IRIS Performer.
- Chapter 12, “Statistics,” discusses the various kinds of statistics you can collect and display about the performance of your application.

- Chapter 13, “Performance Tuning and Debugging,” explains how to use performance measurement and debugging tools and provides hints for getting maximum performance.
- Chapter 14, “Programming with C++,” discusses the differences between using the C and C++ programming interfaces.
- Appendix A, “Image Gallery,” contains some sample images created by using IRIS Performer to display various scene databases.

Conventions

This guide uses the following typographical conventions:

Bold	is used for function names, with parentheses appended to the name. Also, bold lowercase letters represent vectors, and bold uppercase letters denote matrices.
Italics	indicates filenames, IRIX command names, command-line option flags, variables, and book titles.
`Fixed-width`	is used for code examples and system output.
`Bold Fixed-width`	indicates user input, items that you should type in from the keyboard.

Note that in some cases it’s convenient to refer to a group of similarly named IRIS Performer functions by a single name; in such cases an asterisk is used to indicate all the functions whose names start the same way. For instance, **pfNew*()** refers to all functions whose names begin with “pfNew”: **pfNewChan()**, **pfNewDCS()**, **pfNewESky()**, **pfNewGeode()**, and so on.

Most code examples in this guide are written in C; some are in C++. All code examples are available in both C and C++ forms in the source directory */usr/share/Performer/src/pguide*.

Bibliography

You should be familiar with most of the concepts presented in the first few books listed here—notably *Computer Graphics: Principles and Practice* and the IRIS GL or OpenGL books—to make the best use of IRIS Performer and this programming guide. Most of the other books listed here, however, delve into more advanced topics and are listed as further reading for those interested. Information is also provided on electronic access to Silicon Graphics' files containing answers to frequently asked IRIS Performer questions.

Computer Graphics

For a general treatment of a wide variety of graphics-related topics, see:

- Foley, J.D., A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*, 2nd Ed. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1990.
- Newman, W.M., and R.F. Sproull, *Principles of Interactive Computer Graphics*, 2nd Ed. New York: McGraw-Hill, Inc., 1979.

For specific topics of interest to developers using IRIS Performer, also see:

- Akeley, Kurt, "RealityEngine Graphics," *Computer Graphics Annual Conference Series (SIGGRAPH)*, 1993. pp. 309-318.
- Rohlf, John and James Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH)*, 1994, pp. 381-394.
- Helman, James, Sharon Clay, Wes Hoffman, Eric Johnston, Michael Jones, Michael Limber, and Philippe Tarbouriech, "Designing Real-Time 3D Graphics for Entertainment," *Course Notes of 1995 SIGGRAPH Course #6*, 1995.
- Shoemake, Ken. "Animating Rotation with Quaternion Curves," *SIGGRAPH '85 Conference Proceedings* Vol 19, Number 3, 1985.

The IRIS GL and OpenGL Graphics Libraries

For information about IRIS GL, see these Silicon Graphics publications:

- *Graphics Library Programming Guide*, Volumes I and II
- *Graphics Library Programming Tools and Techniques*

To order all three of the above manuals, call 1-800-800-SGI1 (in the U.S. and Canada) and specify part number M4-GLGT-5.2. Outside the U.S. and Canada, please contact your local sales office or distributor.

For information about OpenGL, see:

- Neider, Jackie, Tom Davis, and Mason Woo, *OpenGL Programming Guide*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1993. A comprehensive guide to learning OpenGL.
- OpenGL Architecture Review Board, *OpenGL Reference Manual*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1993. A compilation of OpenGL reference pages.
- *The OpenGL Porting Guide*, a Silicon Graphics publication shipped in IRIS InSight™-viewable on-line format as part of the IRIS Developer Option. Provides information on updating IRIS GL-based software to use OpenGL.

X, Xt, IRIS IM, and Window Systems

In conjunction with OpenGL, you may wish to learn about the X window system, the Xt Toolkit Intrinsics library, and IRIS IM (though note that if you use IRIS Performer's pfWindow routines, windows are handled for you; in that case you don't need to know about any of these topics). For information on X, Xt, and Motif, see the O'Reilly X Window System Series, Volumes 1,2, 4, and 5 (usually referred to simply as "O'Reilly" with a volume number):

- Nye, Adrian, Volume One: Xlib Programming Manual. Sebastopol, California: O'Reilly & Associates, Inc., 1991.
- Volume Two: Xlib Reference Manual, published by O'Reilly & Associates, Inc., Sebastopol, California.
- Volume Four: X Toolkit Intrinsics Programming Manual, by Adrian Nye and Tim O'Reilly, published by O'Reilly & Associates, Inc., Sebastopol, California.
- Volume Five: X Toolkit Intrinsics Reference Manual, published by O'Reilly & Associates, Inc., Sebastopol, California.

For information on IRIS IM, Silicon Graphics' port of OSF/Motif, and on making your application interact well with the Silicon Graphics desktop, see these Silicon Graphics publications:

- *IRIS IM Programming Guide*
- *Indigo Magic User Interface Guidelines*
- *Indigo Magic Desktop Integration Guide*

All three of these books are shipped in IRIS InSight™-viewable on-line format as part of the IRIS Developer Option.

Visual Simulation

For information about visual simulation and the use of simulation systems in training and research, see:

- Rolfe, J.M., and K.J. Staples, eds. *Flight Simulation*. Cambridge: Cambridge University Press, 1986. Provides a comprehensive overview of visual simulation from the basic equations of motion to the design of simulator cabs, optical and display systems, motion bases, and instructor / operator stations. Also includes a historical overview and an extensive bibliography of visual simulation and aerodynamic simulation references.
- Rougelot, Rodney S. "The General Electric Computer Color TV Display," in Faiman, M., and J. Nievergelt, eds. *Pertinent Concepts in Computer Graphics*. Urbana, Ill.:University of Illinois Press, 1969, pp. 261-281. This extensive report gives an excellent overview of the origins of visual simulation. It shows many screen images of the original systems developed for various NASA programs and includes the first real-time textured image. This article provides the basis for understanding the historical development of computer image generation and real-time graphics.
- Schacter, Bruce J., ed. *Computer Image Generation*. New York: John Wiley & Sons, Inc., 1983. Reviews the computer image generation process and provides a detailed analysis of early approaches to system design and implementation. The bibliography refers to early papers by the designers of the first image-generation systems.

Mathematics of Flight Simulation

Stevens, Brian L., and Frank L. Lewis. *Aircraft Control and Simulation*. New York: John Wiley & Sons, Inc., 1992. This book describes the complete implementation of a flight-dynamics model for the F-16 fighter aircraft. It provides the basic equations of motion and explains how the more complex issues are handled in practice. Some source code, in FORTRAN, is included.

Virtual Reality

Kalawsky, Roy S. *Science of Virtual Reality and Virtual Environments*. Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1993.

Geometric Reasoning

These two books address geometric reasoning in general, rather than any specifically computer-related or Performer-specific topics:

- Abbott, Edwin A. *Flatland: A Romance of Many Dimensions*, 6th Ed. New York: Dover Publications, Inc., 1952. The story of A. Square and his journeys among the dimensions.
- Polya, George. *How to Solve It: A New Aspect of Mathematical Method*, 2nd Ed. Princeton, NJ: Princeton University Press, 1973.

Conference Proceedings

The proceedings of the I/ITSEC (Interservice/Industry Training, Simulation, and Education Conference) are a primary source of published visual simulation experience. In the past this conference has been known as the National Training Equipment Center/Industry Conference (NTEC/IC) and the Interservice/Industry Training Equipment Conference (I/ITEC). Proceedings are available from the National Technical Information Service (NTIS). Here are NTIS order numbers for several of the older proceedings:

- Seventh N/IC, November 1974: AD-A000-970 NTEC
- Eighth N/IC, November 1975: AD-A028-885 NTEC
- Ninth N/IC, November 1976: AD-A031-447 NTEC
- Tenth N/IC, November 1977: AD-A047-905 NTEC
- Eleventh N/IC, November 1978: AD-A061-381 NTEC
- First I/ITEC, November 1979: AD-A077-656 NTEC
- Third I/ITEC, November 1981: AD-A109-443 NTEC

The IMAGE Society is dedicated solely to the advancement of visual simulation technology and its applications. It holds conferences and workshops, the proceedings of which are an excellent source of advice and guidance for visual simulation developers. The society can be reached through electronic mail at image@acvax.inre.asu.edu. Some of the IMAGE proceedings published by the Air Force Human Resources Lab AFHRL at Williams AFB prior to the formation of the IMAGE Society are also available from the NTIS. Order numbers are:

- IMAGE, May, 1977: AD-A044-582 AFHRL
- IMAGE II (closing), July, 1981: AD-A104-676 AFHRL
- IMAGE II (proceedings), November, 1981: AD-A110-226 AFHRL

The Society of Photo-Optical Instrumentation Engineers (SPIE) also has articles of interest to visual simulation developers in their conference proceedings. Some of the interesting publications are:

- Vol. 17, *Photo-Optical Techniques in Simulators*, April, 1969
- Vol. 59, *Simulators & Simulation*, March, 1975
- Vol. 162, *Visual Simulation & Image Realism*, August, 1978

Survey Articles in Magazines

- *Aviation Week & Space Technology*, January 17, 1983. Special issue on visual simulation.
- Fischetti, Mark A., and Carol Truxal. “Simulating the Right Stuff.” *IEEE Spectrum*, March, 1985, pp. 38-47.
- Schacter, Bruce. “Computer Image Generation for Flight Simulation.” *IEEE Computer Graphics & Applications*, October, 1981, pp. 29-68.
- Schacter, Bruce, and Narendra Ahuja. “A History of Visual Flight Simulation.” *Computer Graphics World*, May, 1980, pp. 16-31.
- Tucker, Jonathan B., “Visual Simulation Takes Flight.” *High Technology Magazine*, December, 1984, pp. 34-47.

Internet Resources

Answers to common questions.

- Silicon Graphics maintains a publicly accessible directory of questions that developers often ask about IRIS Performer, along with answers to those questions. Each question-and-answer pair is provided in a file of its own, named by topic. To obtain any of these files, use anonymous *ftp* to connect to sgigate.sgi.com; then *cd* to the directory */pub/Performer/selected-topics* and use *ls* to see a list of available topics. Alternatively, use a World Wide Web browser to look at ftp://sgigate.sgi.com/pub/Performer/selected-topics.

Electronic forum for discussions about IRIS Performer.

- The *info-performer* mailing list provides a forum for discussion of IRIS Performer including technical and non-technical issues. Subscription requests should be sent to info-performer-request@sgi.com. Much like the comp.sys.sgi.* newsgroups on the Internet, it isn't an official support channel but is monitored by several interested SGI employees familiar with the toolkit.

World Wide Web page.

- Silicon Graphics maintains a public Web page for IRIS Performer at http://www.sgi.com/Technology/Performer.html. This page can be accessed using a Web browser such as Netscape™. This guide, errata and amendments may be made available through this page.

Chapter 1

“Getting Acquainted With IRIS Performer”

This chapter introduces you to some of the sample IRIS Performer applications and databases.

Chapter 1

Getting Acquainted With IRIS Performer

A complete sample application, *perfly*, as well as many short example programs are included with IRIS Performer. This chapter explains how to use these applications so you can get acquainted with some of the features of IRIS Performer. If you're already somewhat familiar with IRIS Performer or visual simulation in general, you might want to skip ahead to the overview in Chapter 2, "IRIS Performer Basics," or jump directly to whatever chapter covers the specific topic you're interested in.

Source code for *perfly* is provided beneath */usr/share/Performer/src/sample* / C in the *perfly* and *common* directories so that you can incorporate parts of these programs into your own applications. (A C++ version can be found under */usr/share/Performer/src/sample* / C++).

Exploring the IRIS Performer Sample Scenes

The *perfly* application is a fairly basic visual simulation program that can load and store scene databases in any of a wide variety of common formats and display the resulting scenes. This section describes how to use *perfly* to look at several sample databases provided with IRIS Performer.

perfly provides a graphical user interface with which you can control many of the visual simulation features that are described in this guide, such as time-of-day selection, haze density, and so on. These options all default to reasonable values, so you don't need to learn about them before using *perfly*.

Installing the Software

Follow the instructions in the IRIS Performer Release Notes to install the software. This process places the appropriate libraries, header files, sample databases, reference pages, on-line books and demonstration programs on your system.

Figure 1-1 A Section of the New Jerusalem City Hall

Starting and Quitting *perfly*

To launch *perfly*, type

```
IRIS% perfly -d chamber.0.lsa
```

Note: Both OpenGL and IRIS GL versions of *perfly* are installed in */usr/sbin*. The *perfly* command invokes the one more appropriate for your particular graphics hardware. To explicitly invoke the OpenGL or IRIS GL versions, run *perfly_ogl* or *perfly_igl*, respectively.

The *perfly* program allows several motion models; the –d on the command line tells the program to start in the Drive model, which provides an easy way to drive or walk through a scene while maintaining a fixed height above the ground.

When you want to quit *perfly*, either press the **<Esc>** key or click the “Quit” button on the control panel.

Look Around

Look around the scene using the mouse. First, place the cursor in the center of the simulation window. Now depress the middle mouse button and move the mouse to the left and to the right to turn in place; you'll continue to pivot until you place the cursor back in the center of the screen.

Don't despair if you inadvertently start moving around, lose sight of the building, or otherwise lose position or control. Just move the cursor into the control panel area and click the "Reset All" button on the control panel to get back to the original setup.

Approach the Building

To approach the City Hall model, turn until you're facing it (if you aren't already facing it) and then center the mouse in the screen. Depress the left mouse button briefly to start accelerating forward. When you release the button, you'll continue gliding forward at constant speed and can hold down the middle mouse button to steer.

Tap the middle mouse to stop in front of the building (if you actually entered the building, remember the "Reset All" button). Now accelerate backward by pressing the right button. When you're as far back as you want to go, hold down the left mouse button to gradually slow down, or tap the middle mouse button to stop immediately.

Now use the left mouse button again to start moving forward and drive slowly into the model. Notice that the walls closest to you are cut away at first so you can see inside; once you're completely inside the building, those walls reappear. Drive around and explore the building. Tap the middle mouse button to stop before you run into anything (but don't worry—at this point you'll bounce off of any walls you hit). If the walls get in your way, you can turn off collision detection with the button labeled "Collide" on the control panel, or press the **c** key on the keyboard.

More Controls

To see the underlying geometry used to create the model, click the "Style" button in the control panel, or press the **w** key on the keyboard. This changes the display to wireframe mode. In this mode you can more easily see how

many polygons are used to represent an object. This information can be helpful when you're tuning a database, because it's important to know when the number of polygons becomes a limiting factor. To turn wireframe mode off, just click the "Style" button (or press **w**) again. The **W** key can be used to cycle through several different draw styles.

To close the entire control panel (and devote the entire screen to the model), click the "GUI Off" button at the upper right of the control panel, or just press the **<F1>** key. Press the **<F1>** key again to restore the control panel.

If you click the "Stats" button in the control panel, a transparent panel showing scene statistics appears overlaid on the screen. The buttons next to the "Stats" button allow you to choose one of the available statistical displays. Try moving around in the scene while watching how the statistics change. Note in particular that the number of triangles being considered for rendering changes drastically depending on where you look; this demonstrates IRIS Performer's use of *culling* to ignore objects that are completely outside the field of vision. For more information about culling, see Chapter 6, "Database Traversal." For more detailed information on the statistics panels, see Chapter 12, "Statistics."

Several other keys on the keyboard are active and can be used to control *perfly* even when the control panel isn't displayed. Pressing the "**?**" key will print a list of most key sequences to the shell window. For full details, the file */usr/share/Performer/src/sample/perfly/C/keybd.c* contains a list of these key sequences and their effects. Remember that you can always use the **<Esc>** key to terminate *perfly* at any time.

The control panel's field-of-view slider can be used to select a wide angle view, up to 100 degrees.

As you travel through the building, try turning on the fog effect by clicking the Fog button. Experiment with the fog density and other controls. (Remember: If you've closed the control panel, the **<F1>** key will restore it.)

Flying

The Fly motion model provides an alternative to Drive. This model allows full motion in three dimensions (unlike the Drive model, which doesn't allow vertical motion). The mouse in the Fly model is used in much the same way to control motion, but when steering the vertical position of the mouse

in the window controls your vertical tilt. You can select this mode by pressing the right mouse button on the button marked “Drive” and select “Fly” from the menu.

As when driving, the left button makes you go forward and the right button makes you go backward. As long as either button is pressed you’ll continue to accelerate.

You turn holding the middle mouse button down and moving the cursor away from the center of the simulation window. Moving the cursor left or right causes left and right turns, respectively. Moving the cursor up or down causes the view direction to tilt up or down, respectively. The rate of turning and tilting is scaled by the distance of the cursor from the center of the simulation window; that is, no change of direction occurs when the cursor is at the center and full-speed rotation occurs at the edges of the window.

If you want to maintain a steady velocity rather than accelerating, hold down the middle mouse button to steer while using the left and right buttons to control the speed. To stop tap the middle mouse button.

Trackball

The trackball motion model provides a third option for controlling motion. You can select this mode by pressing the right mouse button on the button marked “Fly” and selecting “Trackball” from the menu.

In trackball mode, when you drag with the middle button the object rotates about its center as if it were attached to a large trackball that fills the screen. That is, dragging up and down causes rotation about the horizontal axis parallel to the screen and dragging left and right causes rotation about the vertical axis parallel to the screen.

By dragging with the left mouse, you can translate the object in the direction you drag, left, right, up or down. By dragging with the right mouse, you can translate the object in and out of the screen. In all cases, if you release the mouse button while dragging, the motion continues on its own.

Motion Using Paths

There are other approaches to traveling through a scene than the models described here. You can, for instance, build a specific path into the viewer, to prevent the user from straying outside your model. The path model is supported by a general path-following system in the *libpfutil* library. Many simulation applications require path support for such objects as: cars, trucks, and people (in driver-training software); waiting aircraft both on the ground and in the air (in flight simulation); and opposing forces in military trainers. Path support in *libpfutil* allows paths of varying speeds to be built from line segments and arcs with automatic fillet construction between segments for smooth transitions.

Instances

The bench objects in the City Hall scene were designed using the database concept known as instancing, in which a single geometric object such as a tree, house, car, or, in this case, bench, is used multiple times within a database at different locations and with different positions or scale factors. (In this case, the instances have been *flattened* to improve performance; each bench is now a separate object.) See “Instancing” in Chapter 5 for information on this topic.

Model Implementation and Database-Format Loaders

An important point about the implementation of this model is that the scene was created using a database modeler and is still in its original format. This demonstrates the *data fusion* capability of IRIS Performer: Databases need not be converted to a standard file format before being read in to an IRIS Performer application. Rather, unique file readers are constructed for each format to be used. IRIS Performer can thus work with data from multiple sources concurrently, using a common software interface, without needing intermediate translation or conversion steps.

Because these database importers operate at run time, they’re able to supply executable functions that extend the regular IRIS Performer processing to whatever is required by various formats. This eliminates the *least common denominator* limitation, where database constructs that can’t be represented in the common format must be either deleted or reconstructed. Such limitations are often encountered when all data must be converted to a fixed,

shared file format. In the IRIS Performer paradigm, each importer can provide new functions to support the required features that aren't present in IRIS Performer—extending full native support for each desired database format.

To use the visual interface from *perfly* with your own databases, simply list your databases on the *perfly* command line. *perfly* examines the extension part of each filename to determine what format the file is in. File importers are provided as examples with IRIS Performer (and supported by *perfly*) for many file formats, including 3DStudio "3ds", Designer's Workbench "*dwb*", Medit Productions "*medit*", MultiGen OpenFlight "*flt*", SGI BIN and SGO ("*bin*" and "*sgo*", respectively), Wavefront "*obj*", Open Inventor™ "*iv*", the *STL* stereolithography format, and so on.

If your files are in one or more of these formats, the *perfly* program can be used to view them. If you have files in other formats, you will need to add an importer for those formats to *perfly* before you can use it to view your files. This is not a difficult or daunting task, but it does require an understanding of most of IRIS Performer to be done properly. It should not be undertaken until you have completed reading this book. When you want to develop such file-importing routines, refer to Chapter 9, "Importing Databases," for instructions and a discussion of the provided sample file-importers.

Exploring Code

Being a complete application makes *perfly* a good demonstration for IRIS Performer in action, but it is also a large rather complex piece of code. A better place to start exploring programming with IRIS Performer is the sample code provided in */usr/share/Performer/src/sample/pguide*. Under this directory, you can find examples for programming many of the features available in each of the libraries that make up IRIS Performer, using either C or C++.

Going Beyond Visual Simulation

In *perfly*, you can view an object or scene from any angle and location, from points either inside or outside of the scene. This is the part of the visual simulation development task that IRIS Performer helps you create—the visual part, what you see when you look out the window.

But there's more to a simulation of reality than just visuals. Purely visual simulations of travel have much the same feel whether the simulation is of a boat, a car, a plane, or a magic carpet. In such simulations there's no nonvisual sensory input at all; the user simply watches scenery move past. IRIS Performer leaves the nonvisual aspects—the feel of the simulation—up to you. You determine the vehicle dynamics and construct an apparatus or create code to mimic its behavior. You develop a method for manifesting the physical sensation of how your simulation relates to its environment and responds to stimuli.

When you integrate physical aspects of a simulator with the real-time visuals created with IRIS Performer, the result can be a complete sensory environment, both visual and physical—creating a convincing simulation of reality. Since IRIS Performer puts the tools for rapid development of real-time visuals into your hands, you can spend more time developing the physical part of the simulation.

Another aspect of IRIS Performer that lies below the surface of the demos is its ability to accelerate graphics to top-rated performance levels on Silicon Graphics hardware. This means that IRIS Performer puts a virtual Silicon Graphics hardware and software expert at your fingertips, providing all the tools you need to custom-tune your graphics application for maximum performance on your system.

Deciding Where to Start

Now that you've seen what IRIS Performer can do, it's up to you to decide how to proceed. You can read through the remainder of this programming guide, beginning with the overview in Chapter 2, or you can skip directly to a chapter that describes a specific task you need to perform.

Chapter 2

“IRIS Performer Basics”

This chapter provides a brief overview of IRIS Performer’s features and component libraries.

Chapter 2

IRIS Performer Basics

This chapter provides background for and an introduction to IRIS Performer, including a survey of visual simulation techniques, descriptions of features and libraries, and discussion of some of the specific details of IRIS Performer structure and use.

What Is IRIS Performer?

IRIS Performer is an extensible software toolkit for creating real-time 3D graphics. Typical applications are in the fields of visual simulation, entertainment, virtual reality, broadcast video, and computer aided design. IRIS Performer provides a flexible, intuitive, toolkit-based solution for developers who want to optimize performance on Silicon Graphics systems.

The main components of the toolkit are four *dynamic shared objects* (DSOs), as shown in Table 2-1, support files for those libraries (such as the header files), and source code for sample applications.

Table 2-1 IRIS Performer Libraries

DSO Name	Header File	Description
libpf.so	pf.h	Main IRIS Performer library. Contains *libpf*, which handles multiprocessed database traversal and rendering, and *libpr*, which performs the optimized rendering, state control, and other functions fundamental to real-time graphics.
libpfdu.so	pfdu.h	Library of scene and geometry building tools which greatly facilitate the construction of database loaders and converts. Tools include a sophisticated triangle mesher and state sharing for high performance databases.

Table 2-1 (continued) IRIS Performer Libraries

DSO Name	Header File	Description
libpfutil.so	pfutil.h	Utility-functions library.
libpfui.so	pfui.h	User interface library.

Note that while this document refers often to the *libpr* library or *libpr* "objects", the library itself does not exist in isolation—it has been placed within the *libpf* library to improve instruction-space layout and caching behavior. However, *libpr* still provides an implementation and portability abstraction layer that simplifies the following discussions.

In addition to the core libraries, IRIS Performer provides a suite of database loaders in the form of dynamic shared objects. Each loader reads data files or streams structured in a particular format and converts it into an IRIS Performer scene graph. Loader libraries are named after their corresponding file extension, for example, the Wavefront "obj" format loader is found in *libpfobj.so*. Any number of file loaders may be accessed through the single **pfdLoadFile()** function which uses special dynamic shared object features to locate and use the proper loader corresponding to the extension of the file being loaded.

Figure 2-1 illustrates the relationships between the IRIS Performer libraries and the IRIS system software. All IRIS Performer features are provided as a layer above the IRIX operating system and the graphics library. IRIS Performer doesn't isolate application programs from IRIX or the graphics library, however. Even when using IRIS Performer to its fullest extent, applications have free access to all system layers—including not only *libpf*, *libpr*, and the *libpfdu* loader and builder libraries, but the graphics library and IRIX as well.

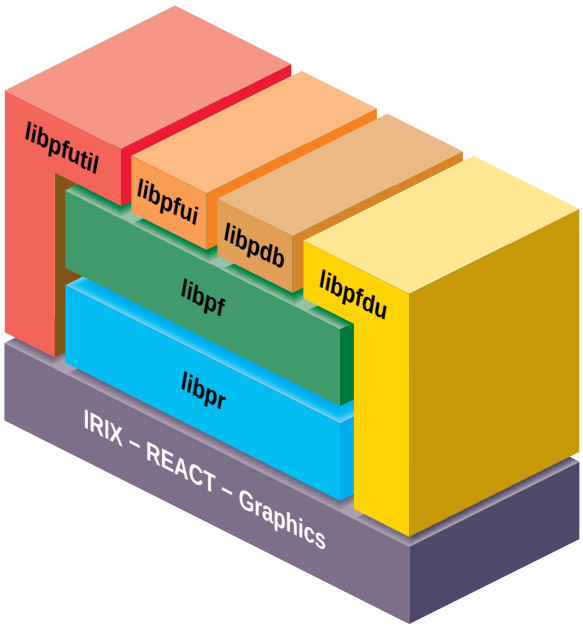


Figure 2-1 IRIS Performer Library Hierarchy

Developers are free to choose which of the libraries best suits their needs. You may want to build your own toolkits on top of *libpr* (but you still link with *libpf*, you just don't use any *libpf* features), or you can take advantage of the visual simulation development environment that *libpf* provides.

IRIS Performer defines a run-time-only database through its programming interface; it doesn't define an archival database or file format. Applications import their databases into IRIS Performer run-time structures. You can either write your own routines to do this or use one of the many database loaders provided as sample source code. These examples show how to import more than thirty popular database formats and how to export scene graphs in the open Designer's Workbench and Medit formats (see Chapter 9 for more information).

This guide describes IRIS Performer in a top-down fashion. Chapters 3 through 8 describe *libpf*, the visual simulation application development library. Chapters 10 and 11 describe *libpr*, the high-performance rendering library. Chapter 9 discusses *libpfdu*, the database utility library and *libpfdb*, the format-specific collection of file load, convert, and store utilities.

Applications

IRIS Performer can be used in a variety of ways. You can use it as a complete database processing and rendering system for applications such as flight simulation, driver training, or virtual reality. It can also be used in conjunction with layered application-development tools to perform the low-level portion of the visual simulation development projects. In short, applications can use part or all of the features provided by IRIS Performer.

For example, consider a driver training application that has already been developed. This application consists of a database, simulation code, and rendering code. The application can be ported to IRIS Performer in several ways. If time is short and the bottleneck is in the rendering code, IRIS Performer's *libpr* library layer can take over the rendering task with minimal effort. Alternatively, it may be better to create an importer to import the existing database into IRIS Performer's run-time format and gain the extra features that *libpf* provides.

Features

This section lists the features of the libraries.

High-Performance Rendering Library (*libpr*) Features

libpr consists of a number of facilities generally required in most visual simulation and real-time graphics applications:

- High-speed geometry rendering functions
- Efficient graphics state management
- Comprehensive lighting and texturing
- Simplified window creation and management.

- Immediate mode graphics
- Display list graphics
- Integrated 2D and 3D text display functions
- A comprehensive set of math routines
- Intersection detection and reporting
- Color table utilities
- Asynchronous filesystem I/O
- Shared memory allocation facilities
- High-resolution clocks and video-interval counters

Visual Simulation Application Library (*libpf*) Features

- Multiple graphics pipeline capability
- Multiple windows per graphics pipeline
- Multiple display channels per window
- Hierarchical scene graph construction and real-time editing
- Multiprocessing (parallel simulation, intersection, cull, and draw processes, and asynchronous database management)
- System stress and load management
- Level-of-detail model switching, with fading
- Rapid culling to the viewing frustum
- Intersections and database queries
- Dynamic and static coordinate systems
- Fixed-frame-rate capability
- Shadows and spotlights
- Morphing
- Visual simulation features
  - An environmental model
  - Light points

  - Animation sequences
  - Sophisticated fog and haze control
  - Landing light capabilities
  - Billboarded geometry

Geometry Builder Library (*libpfdu*) Features

- Allows input in immediate mode fashion, simplifying database conversion
- Produces optimized IRIS Performer data structures
  - Tessellates input polygons including concave and recombines triangles into high performance meshes.
  - Automatically shares state structures between geometry when possible.
  - Produces scene graph containing optimized pfGeoSets and pfGeoStates

Utility Library (*libpfutil*) Features

- Processor isolation routines
- GLX mixed mode utilities
- Device input and event handling, both IRIS GL and X versions
- Cursor control
- Simple and efficient GUI and widgets
- Scene graph traversal utilities
- Texture animation or “movies”
- Smoke and fire effect simulation

User Interface Library (*libpfui*) Features

- Trackball, fly, and drive motion models
- Collision models

Survey of Visual Simulation Techniques

Computers have generated interactive simulated virtual environments—usually for training or entertainment—since the 1960s. Computer image generation (CIG) has not always been a readily available technique, and many special purpose approaches to visual simulation have been tried. For example, the NASA Kennedy Space Center newspaper *Spaceport News* described the Apollo 7 astronaut training visual simulator this way on March 28, 1968:

> Each simulator consists of an instructor's station, crew station, computer complex, and projectors to simulate the stages of a flight. Engineers serve as instructors, instruments keeping them informed at all times of what the pilot is doing. Through the windows, infinity optics equipment duplicates the scenery of space. The main components of a typical visual display for each window includes a 71-centimeter fiber-plastic celestial sphere embedded with 966 ball bearings of various sizes to represent the stars from the first through fifth magnitudes, a mission-effects projector to provide earth and lunar scenes, and a rendezvous and docking projector which functions as a realistic target during maneuvers.[1]

Since such beginnings, the sense of reality that visual simulation systems can provide has advanced significantly, due both to advances in hardware and software and to a greater understanding of human perceptions.

[1] In recognition of the ingenuity of this system, IRIS Performer includes a star database with the locations and magnitudes of the 3010 brightest stars as seen from earth. View the file "*/usr/share/Performer/data/3010.star*" with *perfly* while contemplating the engineering effort required to accurately embed those 966 ball bearings.

This section outlines the major requirements of current visual simulation systems. These requirements fall into six major groups, each covering several related topics:

- Low latency image generation

  Reducing perceived *latency* (the time between input and response) requires reducing both actual latency and frame rate. You cannot avoid latency, but you can minimize its effects by attention to hardware design and software structure.

- Consistent frame rates

  A fixed frame rate is essential to realistic visual simulation. Achieving this goal, however, is very difficult because it requires using a fixed graphics resource to view images of varying complexity. To design for constant frame rates you must understand the required compromises in hardware, database, and application design.

- Rich scene content

  Customers nearly always want complex, detailed, and realistic images, without sacrificing high update rates and low system cost. Thus, providing interesting and natural scenes is usually a matter of tricks and halfway measures; a naive implementation would be prohibitively expensive in terms of machine resources.

- Texture mapping

  Texture processing is arguably the most important incremental capability of real-time image generation systems. Sophisticated texture processing is the factor that most clearly separates the “major league” from the “minor league” in visual simulation technology.

- Real-time character animation

  Real-time character animation in entertainment systems is based on features and capabilities originally developed for high-end flight simulators. Creation of compelling entertainment experiences hinges on the ability to provide engaging synthetic characters.

- Database construction

  One of the key notions of real-time image generation systems is the fact that they are often programmed largely by their databases. This programming includes the design and specification of several autonomous actions for later playback by the visual system.

Low-Latency Image Generation

The issue of latency is critical to comfortable perception of moving images under interactive control. In the real world, the images that reach our brains move smoothly and instantly in reaction to our own motion. In simulated visual environments, such motion is usually depicted as a discrete series of images generated at fixed time intervals. Furthermore, the image resulting from a motion often is not presented until several frame intervals have elapsed, creating a very unnatural latency. A typical human reaction to such delayed images is a nausea commonly known as *simulator sickness*.

In visual simulation the terms “latency” and “transport delay” refer to the time elapsed between stimulus and response. Confusion can enter the picture because there are several important latencies.

The most general measure is the *total latency*, which measures the time between user input (such as a pilot moving a control) and the display of a new image computed using that input. For example, if the pilot of a flight simulator initiates a sudden roll after smooth level flight, how long does it take for a tilted horizon to appear?

The total time required is the sum of latencies of components within the processing path of the simulation system. The basic component latencies include the time required for each of these tasks:

- Input device measurement and reporting
- Vehicle dynamics computation
- Image generation computation
- Video display system scan-out

The latency that matters to the user of the system is the total time delay. This overall latency controls the sense of realness the system can provide.

Another measure combines the latencies due to image generation and video display into the *visual latency*. Questions of latency in visual simulation applications usually refer to either total latency or visual latency. The application developer selects the scope of the application, and then the latency is decided by the choice of image generation mode, frame rate, and video output format.

In many situations the perceived latency can be much less than the actual latency. This is because the human perception of latency can be reduced by anticipating the user's inputs. This means that reducing perceived latency is largely a matter of accurate prediction.

Consistent Frame Rates

To be acceptable by human observers, interactive graphics applications, and immersive virtual environments in particular depend on a consistent frame rate. Human perceptions are attuned to continuous update from natural scenes but seem tolerant of discrete images presented at rates above 15 frames per second—as long as the frame rate is consistent. When latency grows large or frame rates waver, headaches and nausea often result.

Attaining a constant frame rate for a constant scene is easy. What's hard is maintaining a constant frame rate through wildly varying scene content and complexity. Designers of image generation systems use several approaches to achieve a constant, programmer-selected, frame rate.

The first and most basic method is to draw all scenes in such a simple way that they can be viewed from any location without altering the chosen frame rate. This conservative approach is much like always driving in low gear just in case a hill might be encountered. Implementing it simply means identifying and planning for the worst case situation of graphics load. Although this may be reasonable in some cases, in general it's unacceptably wasteful of system resources.

A second approach is to discard (*cull*) database objects that are positioned completely outside the *viewing frustum*. This requires a pass through the visual database to compare scene geometry with the current frame's viewing volume. Any objects completely outside the frustum can be safely discarded. Testing and culling a complex object requires less time than drawing it.

When simple view-volume culling is insufficient to keep scene complexity constant, it may be necessary to compute potential visibility of each object during the culling process by considering other objects within the scene that may occult the test object. High performance image generation systems use comparable occlusion culling tests to reduce the polygon filling complexity of real-time scenes.

Rich Scene Content

Several tricks and techniques can give the impression of rich scene content without actually requiring large quantities of complex geometry.

Level of Detail Selection

Graphics systems can display only a finite number of geometric primitives per frame at a specified frame rate. Because of these limitations, the fundamental problem of database construction for real-time simulation is to maximize visual cues and minimize scene complexity. With *level of detail* selection, one of several similar models of varying complexity is displayed based on how visible the object is from the eyepoint. Level of detail selection is one of the best tools available for improving display performance by reducing database complexity. For more detailed information, see "Level-of-Detail Management" in Chapter 7.

Billboard Objects

Many of the objects in databases can be considered to have one or more axes of symmetry. Trees, for example, tend to look nearly the same from all horizontal directions of view. An effective approach to drawing such objects with less graphic complexity is to place a texture image of the object onto a single polygon and then rotate the polygon during simulation to face the observer. These self-orienting objects are commonly called *billboards.* For information on billboards, see "pfBillboard Nodes" in Chapter 5.

Animation Sequences

Animated events in simulation environments often have a sequence of stages that follow each other without variation. Where this is the case, you can often define this behavior in the database during database construction and allow the behavior to be implemented by the real-time visual system without intervention by the application process.

An example of this would be illuminated traffic signals in a driving simulator database. There are three mutually exclusive states of the signal, one with a green lamp, one with the amber, and one with the red. The duration of each state is known and can be recorded in the database. With these intervals built into the database, simulations can be performed without

requiring the simulation application to cycle the traffic signal from one state to the next.

The simplest type of animation sequence is known as a *geometry movie*. It is a sequence of exclusive objects that are selected for display based on elapsed time from a trigger event. Advancement is tied to frames rather than time, or is based on specific events within the database.

For further information on animation, see "pfSequence Nodes" in Chapter 5.

Antialiasing

Antialiased image generation can have a significant effect on image quality in visual simulation. The difference, though subtle in some cases, has very significant effects on the sense of reality and the suitability of simulators for training. Military simulators often center on the goal of detecting and recognizing small objects on the horizon. Aliased graphics systems produce a "sparkle" or "twinkle" effect when drawing small objects. This artifact is unacceptable in these training applications since the student will come to subconsciously expect such effects to announce the arrival of an opponent and this unfulfilled expectation can prove fatal.

The idea of antialiasing is for image pixels to represent an average or other convolution of the image fragments within a pixel's area rather than simply be a sample taken at the pixel's center. This idea is easily stated but difficult to implement while maintaining high performance.

The *RealityEngine* antialiasing approach is termed *multisampling*. In this system, each pixel is considered to be composed of multiple subpixels. Multisampling stores a complete set of pixel information for each of the several subpixels. This includes such information as color, transparency, and (most importantly) a Z-buffer value.

Providing multiple independent Z-buffered subpixels (the so-called *sub-pixel Z-buffer*) per image pixel allows opaque polygons to be drawn in an arbitrary order since the subpixel Z-comparison will implement proper visibility testing. Converting the multiple color values that exist within a pixel into a single result can either be done as each fragment is rendered into the multisampling buffer or after all polygons have been rendered. For best visual result, transparent polygons are rendered after all opaque polygons have been drawn.

Texture Mapping

The most powerful incremental feature of image generation systems beyond the initial capability to draw geometry is *texture mapping*, the ability to apply textures to surfaces. These textures consist of synthetic or photographic images that are displayed in place of the surfaces of geometric primitives, to modify their surface appearance, reflectance, or shading properties. For each point on a texture-mapped surface, a corresponding pixel from the texture map is chosen to display instead, giving the appearance of warping the texture into the shape of the object's surface.

Surface Appearance

The most obvious use of texture mapping is to generate the appearance of surface details on geometric objects, without making those details into actual geometry. One valuable and widely used addition to these texture processing features is the concept of partly transparent textures. An example of this is the use of billboards (see "pfBillboard Nodes" in Chapter 5). For instance, to display a tree using textures and billboards, create a texture map of a tree (from a photograph, perhaps), marking the background (any part of the texture which does not show part of the tree) as transparent. Then, using a flat rectangle for the billboard, map the texture to the billboard; the transparent regions in the texture become transparent regions of the billboard, allowing other geometry to show through.

Environment Mapping

You can use textures to simulate reflections (usually in a curved surface) of a 3D environment such as a room by using the viewing vector and the geometry's surface normal to compute each screen pixel's index into the texture image. The texture used for this process, the *environment map*, must contain images of the environment to be reflected.

Sophisticated Shading

You can use the environment mapping technique to implement lighting equations by noting that the environment map image represents the image seen in each direction from a chosen point. Interpreting this image as the illumination reflected from an incident light source as a function of angle, the intensities rather than the colors of the environment map can be used to

scale the colors of objects in the database in order to implement complex lighting models (such as Phong shading) with high performance. You can use this method to provide elaborate lighting environments in systems where per-pixel shading calculations would not otherwise be available.

Projective Texture

You can also use texture mapping to project images such as aircraft landing lights and vehicle headlights into images. These projective texture techniques, when combined with the ability to use Z-buffer contents to texture images, allow the generation of real-time images with true 3D cast shadows.

Character Animation

Some interactive applications include animated characters as well as scenery and objects. Character animation is a complex topic with its own needs and techniques.

Morphing

"Terrain Level of Detail" in Chapter 7 describes the simplest active database methodology for continuous terrain level of detail processing based on interpolation between two elevations for vertices, a process also known as *morphing.* Advanced Onyx/RealityEngine real-time image generation hardware is capable of the interpolation of vertex position, colors, normal vectors, and texture coordinates between two versions of a model using the IRIS Performer pfMorph node. Morphing can be used to fill in motion between a start position and an end position for an object or—in its fully generalized form—parts of an animated character (such as facial expressions).

Generalized Morphing

Simple pair-wise morphing is not sufficient to give animated characters life-like emotional expressions and behavior. You need the ability to model multiple expressions in an orthogonal manner and then combine them with arbitrary weighting factors during real-time simulation.

One current approach to human facial animation is to build a geometric model of an expressionless face, and then to distort this neutral model into an independent target for each desired expression. Examples include faces with frowns and smiles, faces with eye gestures, and faces with eyebrow movement. Subtracting the neutral face from the smile face gives a set of smile displacement vectors and increases efficiency by allowing removal of null displacements. Completing this process for each of the other gestures yields the input needed by a real-time system: a base or neutral model and a collection of displacement vector sets.

In actual use, you would process the data in a straightforward manner. You would specify the weights of each source model (or corresponding displacement vector set) before each frame is begun. For example a particular setting might be “62% grin and 87% arched eyebrows” for a clownish physiognomy. The algorithmic implication is simply a weighted linear combination of the indicated vectors with the base model.

These processing steps are made more complicated in practice by the performance-inspired need to execute the operations in a multiprocessing environment. Parallel processing is needed because users of this technology

- need to perform hundreds to thousands of interpolations per character
- desire several characters in animation simultaneously
- prefer animation update rates of 30 or 60 Hertz
- generate multiple independent displays from a single system

Taken together, these demands can require significant resources, even when only vertex coordinates are interpolated. When colors, normals, and texture coordinates are also interpolated, and especially when shared vertex normals are recomputed, the computational complexity is correspondingly increased.

The computational demands can be reduced when the rate of morphing is less than the image update rate. The quality of the interpolated result can often be improved by applying a non-linear interpolation operation such as the eased cosine curves and splines found useful in other applications of computer animation.

Skeleton Animation

A successful concept in computer-assisted 2D animation systems is the notion of *skeleton animation*. In this method you interpolate a defining skeleton and then position artwork relative to the interpolated skeleton. In essence, the skeleton defines a deformation of the original 2D plane, and the original image is transformed by this mapping to create the interpolated image. This process can be extended directly into the three-dimensional domain of real-time computer image generation systems and used for character animation in entertainment applications.

Total Animation

The techniques of generalized morphing and skeleton animation can be used in conjunction to create advanced entertainment applications with life-like animated characters. One application of the two methods is to first perform a generalized betweening operation that builds a character with the desired pre-planned animation aspects, such as eye or mouth motion, and then to set the matrices or other transformation operators of the skeleton transformation operation to represent hierarchical motions such as those of arms or legs. The result of these animation operations is a freshly posed character ready for display.

Database Construction

Several companies produce database modeling tools that are well integrated with IRIS Performer. A selection of these products are included and described in the Friends of Performer distribution. The Friends of Performer gift software is located in the */usr/share/Performer/friends* directory. These tools have been built to address many aspects of the database construction process. Popular systems include tools that allow interactive design of geometry, easy editing and placement of texture images, flexible file-based instancing, and many other operations. Special-purpose tools also exist to aid in the design of roadways, instrument panels, and terrain surfaces.

The reward of building complex databases that accurately and efficiently represent the desired virtual environment is great, however, since real-time image generation systems are only as good as the environments they're used to explore.

Overview of the IRIS Performer Libraries

This section outlines the basic elements of each library.

The *libpf* Visual Simulation Library

libpf is the visual simulation development library. Functions from *libpf* make calls to *libpr* functions; *libpf* thus provides a high-performance yet easy-to-use interface to the hardware.

Multiprocessing Framework

libpf provides a pipelined multiprocessing model for implementing visual simulation applications. The pipeline stages are

- application
- cull
- draw

The application stage updates and queries the scene. The cull stage traverses the scene and adds all potentially visible geometry to a special *libpr* display list, which is then rendered by the draw stage. Rendering pipelines can be split into separate processes (from one to three) to tailor the application to the number of available CPUs.

In addition, IRIS Performer provides an intersection stage, which may be multiprocessed, that intersects line segments with the database for things like collision detection and line-of-sight determination.

Multiprocess operation is largely transparent because IRIS Performer manages the difficult multiprocessing issues—such as process timing, synchronization, and data coherence—for you.

Pipes, Pipe Windows, and Channels (pfPipe, pfPipeWindow, pfChannel)

libpf provides software constructs to facilitate rendering a visual database. A pfPipe is a rendering pipeline that renders one or more pfChannels into one or more pfPipeWindows. A pfChannel is a view into a visual database—equivalent to a viewport within a pfPipeWindow.

IRIS Performer supports multiple pfChannels on a single pfPipeWindow, multiple pfPipeWindows on a single pfPipe and multiple pfPipes per machine for multichannel, multiwindow, and multipipe operation. Frame synchronization between channels is provided for simulations that display multiple simultaneous views on different hardware displays.

Visual Database (pfScene)

libpf supports a general database hierarchy, defined as a directed acyclic graph of nodes. IRIS Performer provides specialized node types useful for visual simulation applications:

Grouping Nodes

- pfScene: Root node of a visual database
- pfGroup: Branch node, which may have children
- pfSCS: Static coordinate system
- pfDCS: Dynamic coordinate system
- pfLayer: Coplanar geometry node
- pfLOD: Level-of-detail selection node
- pfSwitch: Select among children
- pfSequence: Sequenced animation node
- pfPartition: Collection of geometry organized for efficiency
- pfMorph: Combines (“morphs”) attributes like color, coordinates

Geometric Nodes

- pfGeode: Geometry node
- pfBillboard: Geometry that rotates to face the viewpoint

- pfLightPoint: Luminescent light points
- pfText: Geometry based upon pfFont and pfString
- pfLightSource: User-manipulatable lights which support high-quality spotlights and shadows

A visual database is a graph of nodes that is rooted by a pfScene. A pfScene is viewed by a pfChannel, which in turn is culled and drawn by a pfPipe. Scenes are typically, but not necessarily, constructed by the application at database loading time. IRIS Performer supplies sample source code that shows how to construct a scene from several popular database formats; see Chapter 9 for more information.

IRIS Performer provides traversal functions that act on a pfScene or portions thereof. These functions include

- comprehensive, user-directed intersections
- flattening modeling transformations for improved cull, intersection, and rendering performance
- cloning a database subgraph to obtain model instancing which shares geometry but not articulations
- deletion of scene-graph components
- printing for debugging purposes

The application can direct and customize traversals through the use of identification masks on a per-node basis and through the use of function callbacks.

Frame Control

IRIS Performer is designed to run at a fixed frame rate specified by the application. IRIS Performer measures graphics load, and uses that information to compute a stress value. Stress is applied to the model's level of detail to reduce scene complexity when nearing graphics overload conditions.

IRIS Performer also provides frame synchronization between pfChannels and the Geometry Pipeline™ hardware to ensure that displays are updated in lockstep.

Special Features (pfEarthSky, pfSequence, pfLightPoint)

libpf provides an environmental model called a pfEarthSky, consisting of ground and sky polygons, that efficiently clears the viewport before rendering the scene. Atmospheric effects such as ground fog, haze, and clouds are included.

Sequenced animations, using pfSequence nodes, allow the application to efficiently render complex geometry sequences that aren't easily modeled otherwise. You can think of animation sequences as a series of "flip cards" where the application controls which card is shown and for how long.

Light points, defined by pfLightPoint nodes, can be used to simulate runway lights, approach lights, strobes, beacons, and street lights. The parameters for a pfLightPoint include size, directionality, shape, color, and intensity. IRIS Performer renders light points using modes appropriate to the hardware. Enhanced support for light point simulation is provided by the IRIS Performer pfLPointState mechanism, which substantially replaces the use of pfLightPoint nodes in advanced visual simulation applications.

The *libpr* High-Performance Rendering Library

libpr is a low-level graphics library that supports a variety of functions useful for any high-performance graphics application.

High-Performance Geometry Rendering (pfGeoSet)

Many graphics applications are limited by CPU overhead in sending graphics commands to the Geometry Pipeline. A pfGeoSet is a collection of like primitives such as points, lines, triangles, and triangle strips. pfGeoSets use tuned rendering loops to eliminate the CPU bottleneck.

Efficient Graphics State Management (pfState)

IRIS Performer provides functions to control aspects of graphics library state such as lighting, texture, and transparency. These functions operate in both immediate and *libpr* display-list mode for direct mode changes as well as for mode caching. IRIS Performer manages state changes to optimize graphics library performance.

Other state functions such as push, pop, and override allow extensive control of graphics state.

Graphics State Encapsulation (pfGeoState)

A pfGeoState is an encapsulation of graphics state—a graphics context. Applying a pfGeoState ensures that the graphics pipeline is configured appropriately regardless of previous graphics state. pfGeoStates are very efficient, simplifying and accelerating graphics state management.

Display Lists (pfDispList)

IRIS Performer supports special *libpr* display lists. They don't use graphics library objects, but rather a simple token / data mechanism that doesn't cache geometry data. These display lists cache only *libpr* state and rendering commands. They also support function callbacks to allow applications to perform special processing during display list rendering. Display lists can be reused and are therefore useful for multiprocessing producer / consumer situations in which one process generates a display list of the visible scene while another one renders it. Note that you can also use OpenGL and IRIS GL display lists in IRIS Performer applications.

Math Support

Extensive linear algebra and simple geometric functions are provided. Some supported data types are point, segment, vector, plane, matrix, cylinder, sphere, frustum, and quaternion.

Intersections

Functions are provided to perform intersections of segments with cylinders, spheres, boxes, planes, and geometry. Intersection functions for spheres, cylinders, and frustums are also provided.

Color Tables (pfColortable)

IRIS Performer supports global color tables that can define the colors used by pfGeoSets. Color tables can be used for special effects such as infrared lighting and can be switched in real time.

Asynchronous File I/O (pfFile)

A simple *nonblocking file access* method is provided to allow applications to retrieve file data during real-time operation. This file I/O is very similar to the standard IRIX file I/O functions.

Memory Allocation (pfMalloc(), pfDataPool)

IRIS Performer includes routines to allocate memory from the application process *heap* or from shared memory *arenas*. Shared arenas must be used when multiple processes need to share data. The application can create its own shared memory arenas or use pfDataPools. pfDataPools are shared arenas that can be shared by multiple processes. Applications can allocate blocks of memory within pfDataPools, which can be individually locked and unlocked to provide mutual exclusion between unrelated processes.

High-Resolution and Video-Rate Clocks (pfGetTime, pfVClock)

IRIS Performer includes high-resolution clock and video interval counter routines. **pfGetTime()** returns the current time at the highest resolution that the hardware supports. Processes can either share synchronized time values with other processes or have their own individual clocks.

The video interval counter is tied to the video retrace rate and can synchronize a process with any multiple of the video rate; this mechanism is the basis for producing fixed frame rates.

Intuitive Interface

All the IRIS Performer commands have intuitive names that describe what they do. These mnemonic names make it easy for you to learn and remember the commands. The names may look a little strange to you if you're unfamiliar with this type of interface because they use a mixture of upper- and lowercase letters. Naming conventions provide for consistency and uniqueness, both for routines and symbolic tokens. Following similar naming practices in the software that you develop will make it easier for you and others on your team to understand and debug your code.

Naming conventions for IRIS Performer are as follows:

- All command and token names, whether associated with *libpf* or *libpr*, are preceded by the letters “pf”, denoting the IRIS Performer library.
- Command and type names are mixed-case, while token names are uppercase. For example, “pfTexture” is a type name and “PFTEX_SHARPEN” is a token name.
- In type names, the part following the “pf” is usually spelled out in full—as is the case with “pfTexture”—but in some cases a shortened form of the word is used. For example, “pfDispList” is the name of the display-list type.
- Much of IRIS Performer’s interface involves setting parameters and retrieving parameter values. For the sake of brevity, the word “Set” is omitted from function names, so that instead of “**pfSetMtlColor()**,” “**pfMtlColor()**” is the name of the routine used for setting the color of a pfMaterial. “Get,” however, is not omitted from the names of routines that get information, such as “**pfGetMtlColor()**”.
- Routine names are constructed by appending a type name to an operation name. The operation name always precedes the type name. In this case, the operation name is unabbreviated and the type name is abbreviated. For example, the name of the routine that applies a pfTexture is “**pfApplyTex()**”.

  Compound type names are abbreviated by the first initial of the first word and the entire second word. For example, to draw a display list, which is type pfDispList, use **pfDrawDList()**.
- Symbolic token names incorporate another abbreviation, usually shorter, of the type name. For example:
  - pfTexture tokens begin with “PFTEX_”.
  - pfDispList tokens begin with “PFDL_”.

  This convention ensures that tokens for a particular type have their own name space.
- Other tokens and identifiers follow the conventions of ANSI C and C++ wherein a valid identifier consists of upper and lower-case alphabetic characters, digits, and underscores, and the first character must not be a digit. Further details of these coding conventions as they pertain to C++ programming are described in Chapter 14, “Programming with C++.”

The *libpfdu* Geometry Builder Library

libpfdu is a database-utilities library. It provides helpful functions for constructing optimized IRIS Performer data structures and scene graphs. It's used mainly by database loaders which take an external file format containing 3D geometry and graphics state and load them into IRIS Performer optimized run-time only structures. Such utilities often prove very useful; most modeling tools and file formats represent their data in structures that correspond to the way users model data, but such data structures are often mutually exclusive with effective and efficient IRIS Performer run-time structures.

libpfdu contains many utilities—including DSO support for database loaders and their modes, file path support, and so on—but the heart of *libpfdu* is the notion of the IRIS Performer database builder. The builder is a tool that allows users to input or output a collection of geometry and graphics state in immediate mode.

Users send geometric primitives one at a time, each with its corresponding graphics state, to the builder. When the builder has received all the data, the user simply requests optimized IRIS Performer data structures which can then be used as a part of a scene graph. The builder hashes geometry into different 'bins' based on the geometry's attribute binding types and associated graphics state. It also keeps track of graphics state elements (textures, materials, light models, fog, and so on) and shares state elements whenever possible. Finally, the builder creates pfGeoSets that contain triangle meshes created by running the original geometry through the *libpfdu* triangle-meshing utility.

To go along with each pfGeoSet, the builder creates a pfGeoState (IRIS Performer's encapsulated state primitive) which has been optimized to share as many attributes as possible with other pfGeoStates being built (and possibly with the default pfGeoState that can be attached to a channel with **pfChanGState()**).

Having created all of these primitives (pfGeoSets and pfGeoStates) the builder will place them in a leaf node (pfGeode), and optionally create a spatial hierarchy (for increased culling efficiency) by running the new database through a spatial breakup utility function which is also contained in *libpfdu*. Note that the builder also allows the user to extend the notion of a graphics state by registering callback functionality through builder API and

then treating this state or functionality like any other IRIS Performer state or mode (although such uses of the builder are slightly more complicated). In short, *libpfdu* is a collection of utilities that effectively act as a data funnel where users enter flattened 3D graphics information and are given in return fully functional and optimized IRIS Performer run-time structures.

Current *libpfdu* modules:

- breakup.c
- builder.c
- callbacks.c
- extensors.c
- geobuilder.c
- loadfile.c
- openfile.c
- rebuild.c
- share.c
- spatial.c
- tmesher.c

The *libpfdb* Loader Library

libpfdb is a collection of independent loaders, each of which loads a particular file format into IRIS Performer. Among the formats supported are OpenFlight, Designer's Workbench, Medit, and Wavefront. Each of the *libpfdb* loaders is located in its own source directory; the more complicated loaders may include further source subdirectories. Users usually don't call these functions directly; instead, they call the *libpfdu* function **pfdLoadFile()** which uses the extension part of the file name (the part after the "." character) to decide the format of the file and then automatically invokes the proper loader from *libpfdb*.

Many file loaders have been developed, and most are available in source form as part of the IRIS Performer distribution. Using these loaders as templates, you can write custom loaders for whatever formats you require in your applications.

Database Formats and IRIS Performer

Although IRIS Performer doesn't define a file format, it does provide sample source code for importing numerous other database formats into IRIS Performer's run-time structures. Figure 2-2 shows how databases are imported into IRIS Performer: first a user creates a database with a modeling program, then an IRIS Performer-based application imports that database using one of the many importing routines. IRIS Performer routines then manipulate and draw the database in real time.

Scene graphs can also be generated automatically by loaders with built-in scene-graph generation algorithms. The "sponge" loader is an example of such automatic generation; it builds a model of the Menger (Sierpinski) Sponge, without requiring an input file.

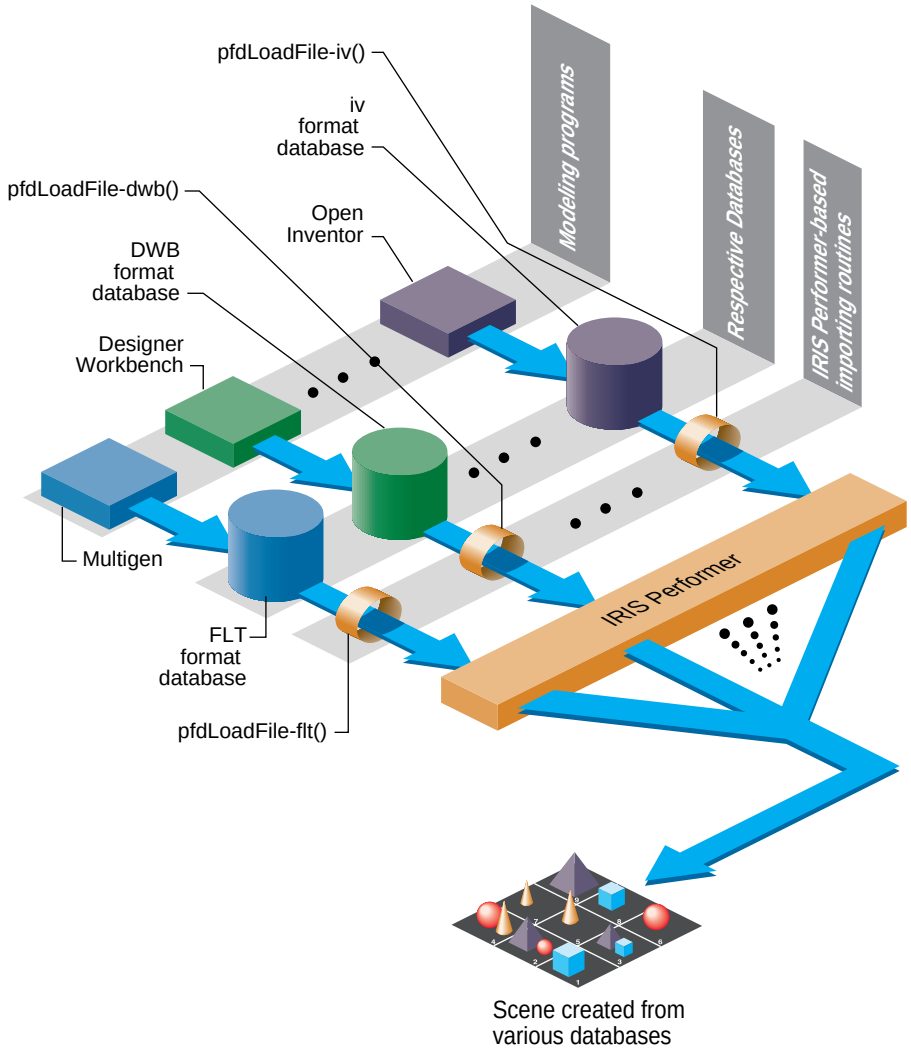


Figure 2-2 Relationship of IRIS Performer to Database Formats

Graphics Libraries

IRIS Performer supports two graphics library APIs: IRIS GL and OpenGL. IRIS GL is supported as the primary graphics API for all systems prior to Indigo2/Impact, such as the RealityEngine; Silicon Graphics recommends using OpenGL on Impact and subsequent graphics hardware, although OpenGL is supported on RealityEngine and many previous systems for application development and porting purposes.

The main difference between IRIS Performer's use of IRIS GL and of OpenGL is in how it handles windows and input; there are relatively few circumstances in which your IRIS Performer-based program will call graphics library routines directly (this will usually only happen in draw callbacks—see "pfNode Cull and Draw Callbacks" in Chapter 6). In fact, most of the differences between IRIS GL and OpenGL are transparent to a developer using IRIS Performer; the choice of which graphics library your application ends up using depends mostly on which libraries you link with at compile time.

The names of the IRIS GL-based libraries for IRIS Performer use the suffix *_igl*; the names of the OpenGL-based libraries end in *_ogl*. For example, to use the OpenGL version of *libpf*, you would link to *libpf_igl.so*.

For information on compiling and linking IRIS Performer applications, see "Compiling and Linking IRIS Performer Applications" in Chapter 3.

OpenGL

OpenGL is a window-system-independent library of graphics functions. Describing OpenGL in detail is beyond the scope of this guide; to learn about OpenGL, see the books mentioned in the "Bibliography" on page xxix.

Porting From IRIS GL to OpenGL

If you have an IRIS Performer application that uses IRIS GL, you can port it to use OpenGL with minimal work. Most of what you need to do is port the window- and event-handling to use X; OpenGL doesn't have any window or event routines. The *OpenGL Porting Guide* provides more information on

porting from IRIS GL to OpenGL, and the sample applications distributed with IRIS Performer provide many examples of programs that compile and run with either IRIS GL or OpenGL.

The pfWindow Windowing Functions

IRIS Performer provides window-system-independent window routines to allow greater portability of applications. You can use these routines whether your application uses IRIS GL windowing, mixed-model IRIS GL / X windowing, or OpenGL with X. For information about these window routines, see the pfWindow(3pf) reference page.

For sample programs involving windows and input handling, see */usr/share/Performer/src/pguide/{libpr,libpf,libpfutil,libpfui}*.

X and IRIS IM

The X Window System is a network-based, hardware-independent window system for use with bitmapped graphics displays. In the X client / server model, an X server running in the background handles input and output, and informs client applications when various events occur. A special client, the *window manager*, places windows on the screen, handles icons, and manages the titles and borders of windows.

IRIS IM is Silicon Graphics' port of OSF / Motif, a set of *widgets* for use with Xt, the X toolkit intrinsics library.

With the pfWindow window functions that IRIS Performer provides, you don't need to know X or IRIS IM to use windows. However, if you want to learn about those topics anyway, see the books mentioned in the "Bibliography" on page xxix.

pfObjects and the Class Hierarchy

IRIS Performer provides an object-oriented programming interface to most of its data structures. Only IRIS Performer functions can change the values of elements of these data structures; for instance, you must call **pfMtlColor()** to set the color of a pfMaterial structure rather than modifying the structure directly. Instances of these "encapsulated" data types are referred to as pfObjects. In addition, some simple data types are not encapsulated for speed and easy access. Examples include pfMatrix and pfSphere. These are referred to as "public" structs or arrays.

In order to allow some functions to apply to multiple data types, IRIS Performer uses the concept of *class inheritance*. Class inheritance takes advantage of the fact that different data types (classes) often share attributes. For example, a pfGroup is a node which can have children. A pfDCS (Dynamic Coordinate System) has the same basic structure as a pfGroup, but also defines a transformation to apply to its children—in other words, the pfDCS data type inherits the attributes of the pfGroup and adds new attributes of its own. This means that all functions that accept a pfGroup* argument will alternatively accept a pfDCS* argument.

For example, **pfAddChild()** takes a pfGroup* argument, but

```
pfDCS *dcs = pfNewDCS();
pfAddChild(dcs, child);
```

appends *child* to the list of children belonging to *dcs*.

Because the C language does not directly express the notion of classes and inheritance, arguments to functions must be cast before being passed, e.g. pfAddChild((pfGroup*)dcs, (pfNode*)child). In the example above, no such casting is required because IRIS Performer provides macros that perform the casting when compiling with ANSI C. (Note, however, that using automatic casting eliminates type checking—the macros will cast anything to the desired type. If you make a mistake and pass an unintended data type to a casting macro, the results may be unexpected.)

No such trickery is required when using the C++ API, so full type checking is always available at compile time.

Inheritance Graph

The relations between classes can be arranged in a directed acyclic inheritance graph in which each child inherits all of its parent's attributes, as illustrated in Figure 2-3. IRIS Performer does not use *multiple inheritance*, so each class has only one parent in the graph.

Note: It's important to remember that an inheritance graph is different from a scene graph. The inheritance graph shows the inheritance of data elements and member functions among user-defined data types; the scene graph shows the relationship among instances of nodes in a hierarchical scene definition.

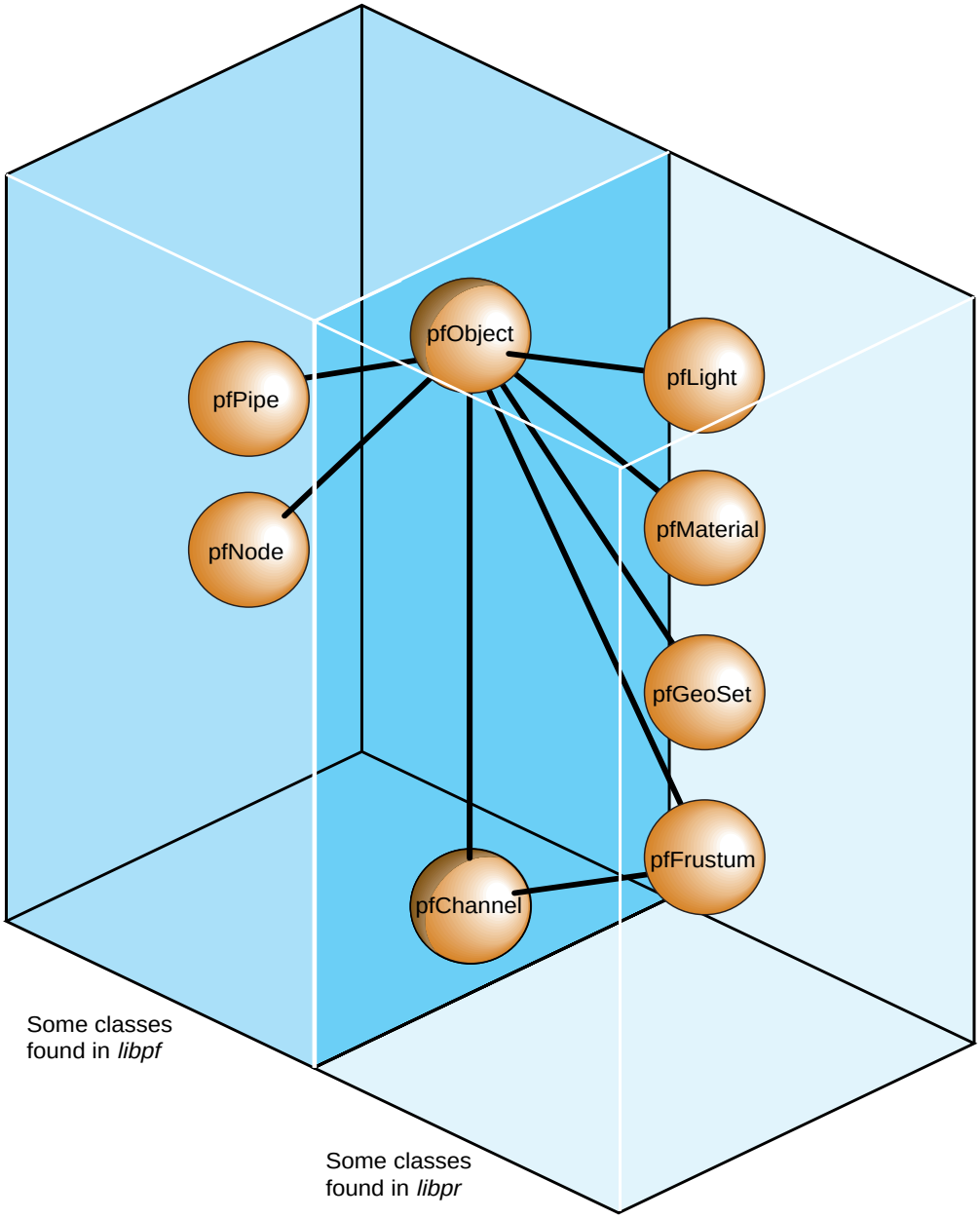


Figure 2-3 Partial Inheritance Graph of IRIS Performer Data Types

IRIS Performer objects are divided into two groups: those found in the *libpf* library and those found in the *libpr* library. These two groups of objects have some common attributes, but also differ in some respects.

While IRIS Performer only uses single inheritance, some objects encapsulate others, hiding the encapsulated object but also providing a functional interface that mimics its original one. For example a pfChannel has a pfFrustum, a pfFrameStats has a pfStats, and a pfPipeWindow has a pfWindow. In these cases, the first object in each pair provides functions corresponding to those of the second. For example, pfFrustum has a routine,

```
pfMakeSimpleFrust(frust, 45.0f);
```

and pfChannel has a corresponding routine,

```
pfMakeSimpleChan(channel, 45.0f);
```

All of the major classes in IRIS Performer are derived from the pfObject class. It unifies the data types by providing common attributes and functions. However, some distinctions remain between *libpr* objects and *libpf* objects, due to memory and performance considerations.

User Data

The primary attribute of a pfObject is called “user data”; its primary functions are **pfDelete()**, **pfCopy()**, and **pfPrint()**.

Each pfObject contains a special pointer to void which is reserved for application use. This pointer can be set with **pfUserData()** to point to a user-supplied data structure. This user structure usually further describes the pfObject by adding attributes that make sense to the application.Both *libpf* and *libpr* objects have user data pointers.

pfDelete() and Reference Counting

Most kinds of data objects in IRIS Performer can be placed in a hierarchical scene graph, using instancing (see “Instancing” in Chapter 5) when an object is referenced multiple times. Scene graphs can become quite complex, which can cause problems if you’re not careful. Deleting objects can be a particularly dangerous operation, for example, if you delete an object that another object still references.

Reference counting provides a bookkeeping mechanism that makes object deletion safe: an object is never deleted if its reference count is greater than zero. Only *libpr* objects have an explicit reference count. In *libpf*, pfNodes are reference-counted by their parents in the scene graph, and all other *libpf* objects (such as pfChannels and pfPipes) have no reference count because you cannot currently delete them.

All *libpr* objects (such as pfGeoState and pfMaterial) have a reference count that specifies how many other objects refer to it. A reference is made whenever an object is attached to another using the IRIS Performer routines shown in Table 2-2.

Table 2-2 Routines that Modify *libpr* Object Reference Counts

Routine	Action
pfGSetGState	Attaches a pfGeoState to a pfGeoSet
pfGStateAttr	Attaches a state structure (such as a pfMaterial) to a pfGeoState
pfGSetHlight	Attaches a pfHighlight to a pfGeoSet
pfTexDetail	Attaches a detail pfTexture to a base pfTexture
pfGSetAttr	Attaches attribute and index arrays to a pfGeoSet
pfTexImage	Attaches an image array to a pfTexture
pfAddGSet, pfReplaceGSet, pfInsertGSet	Modify pfGeoSet / pfGeode association

When object A is attached to object B, the reference count of A is incremented. Additionally, if A replaces a previously referenced object C, then the reference count of C is decremented. Example 2-1 demonstrates how reference counts are incremented and decremented.

Example 2-1 Objects and Reference-Counts

```
pfGeoState *gstateA, *gstateC;
pfGeoSet *gsetB;

/* Attach gstateC to gsetB. Reference count of gstateC
 * is incremented. */
pfGSetGState(gsetB, gstateC);

/* Attach gstateA to gsetB, replacing gstateC. Reference
 * count of gstateC is decremented and that of gstateA
 * is incremented. */
pfGSetGState(gsetB, gstateA);
```

This automatic reference counting done by IRIS Performer routines is usually all you'll ever need. However, the routines **pfRef()**, **pfUnref()**, and **pfGetRef()** allow you to increment, decrement, and retrieve the reference count of a *libpr* object should you wish to do so. (These routines also work with objects allocated by **pfMalloc()**; see "Memory Allocation" in Chapter 10 for more information).

An object whose reference count is less than or equal to 0 can be deleted with **pfDelete()**. **pfDelete()** works for all *libpr* objects and all pfNodes but not for other *libpf* objects like pfPipe and pfChannel. **pfDelete()** first checks the reference count of an object. If the reference count is non-positive, **pfDelete()** decrements the reference count of all objects that the current object references, then it deletes the current object. **pfDelete()** doesn't stop here but continues down all reference chains, deleting objects until it finds one whose count is greater than zero. Once all reference chains have been explored, pfDelete returns a boolean indicating whether it successfully deleted the first object or not. Example 2-2 illustrates the use of **pfDelete()** with *libpr*.

Example 2-2 Using **pfDelete()** with *libpr* Objects

```
pfGeoState *gstate0, *gstate1;
pfMaterial *mtl;
pfGeoSet *gset;

gstate0 = pfNewGState(arena); /* initial ref count is 0 */
```

```
gset = pfNewGSet(arena); /* initial ref count is 0 */
mtl = pfNewMtl(arena); /* initial ref count is 0 */

/* Attach mtl to gstate0. Reference count of mtl is
 * incremented. */
pfGStateAttr(gstate0, PFSTATE_FRONTMTL, mtl);

/* Attach mtl to gstate1. Reference count of mtl is
 * incremented. */
pfGStateAttr(gstate1, PFSTATE_FRONTMTL, mtl);

/* Attach gstate0 to gset. Reference count of gstate0 is
 * incremented. */
pfGSetGState(gset, gstate0);

/* This deletes gset, gstate0, but not mtl since gstate1 is
 * still referencing it. */
pfDelete(gset);
```

Example 2-3 illustrates the use of **pfDelete()** with *libpf*.

Example 2-3 Using **pfDelete()** with *libpf* Objects

```
pfGroup *group;
pfGeode *geode;
pfGeoSet *gset;

group = pfNewGroup(); /* initial parent count is 0 */
geode = pfNewGeode(); /* initial parent count is 0 */
gset = pfNewGSet(arena); /* initial ref count is 0 */

/* Attach geode to group. Parent count of geode is
 * incremented. */
pfAddChild(group, geode);

/* Attach gset to geode. Reference count of gset is
 * incremented. */
pfAddGSet(geode, gset);

/* This has no effect since the parent count of geode is 1.*/
pfDelete(geode);

/* This deletes group, geode, and gset */
pfDelete(group);
```

Some notes about reference counting and **pfDelete()**:

- All reference count modifications are locked so that they guarantee mutual exclusion when multiprocessing.
- Objects added to a pfDispList don't have their counts incremented due to performance considerations.
- In the multiprocessing environment of *libpf*, the successful deletion of a pfNode doesn't have immediate effect but is delayed one or more frames until all processes in all processing pipelines are through with the node. This accounts for the fact that pfDispLists don't reference-count their objects.
- **pfUnrefDelete(obj)** is shorthand for

  ```
  pfUnref(obj);
  pfDelete(obj);
  ```

- Objects whose count reaches zero are not automatically deleted by IRIS Performer. You must specifically request that an object be deleted with **pfDelete()** or **pfUnrefDelete()**.

Copying Objects with pfCopy()

pfCopy() is currently implemented for *libpr* (and **pfMalloc()**) objects only. Object references are copied and reference counts are modified appropriately, as illustrated in Example 2-4.

Example 2-4 Using **pfCopy()**

```
pfGeoState *gstate0, *gstate1;
pfMaterial *mtlA, *mtlB;

gstate0 = pfNewGState(arena);
gstate1 = pfNewGState(arena);
mtlA = pfNewMtl(arena); /* initial ref count is 0 */
mtlB = pfNewMtl(arena); /* initial ref count is 0 */

/* Attach mtlA to gstate0. Reference count of mtlA is
 * incremented. */
pfGStateAttr(gstate0, PFSTATE_FRONTMTL, mtlA);
```

```
/* Attach mtlB to gstate1. Reference count of mtlB is
 * incremented. */
pfGStateAttr(gstate1, PFSTATE_FRONTMTL, mtlB);

/* gstate1 = gstate0. The reference counts of mtlA and mtlB
 * are 2 and 0 respectively. Note that mtlB is NOT deleted
 * even though its reference count is 0. */
pfCopy(gstate1, gstate0);
```

Determining Object Type

Sometimes you have a pointer to a pfObject but you don't know what it really is—is it a pfGeoSet, a pfChannel, or something else? **pfGetType()** returns a pfType which specifies the type of a pfObject. This pfType can be used to determine the class ancestry of the object. Another set of routines, one for each class, returns the pfType corresponding to that class, e.g. **pfGetGroupClassType()** returns the pfType corresponding to pfGroup.

Since pfTypes are most frequently used to answer the question is object A of class B, a helper function **pfIsOfType()** performs the test without the explicit use of a pfType.

With these functions you can test for class type as shown in Example 2-5.

Example 2-5 General-Purpose Scene Graph Traverser

```
void
travGraph(pfNode *node)
{
    if (pfIsOfType(node, pfGetDCSClassType()))
        doSomethingTransforming(node);

    /* If 'node' is derived from pfGroup then recursively
     * traverse its children */
    if (pfIsOfType(node, pfGetGroupClassType()))
        for (i = 0; i < pfGetNumChildren(node); i++)
            travGraph(pfGetChild(node, i));
}
```

Because IRIS Performer allows subclassing of built-in types, when decisions are made based on the type of an object, it is usually better to use **pfIsOfType()** to test the type of an object rather than to test for the strict

equality of the pfTypes. Otherwise the code will not have reasonable default behavior with file loaders or applications which use subclassing.

While the pfType returned from **pfGetType()** is useful to programs, it doesn't mean much to humans. **pfGetTypeName()** returns a null-terminated ASCII string that clearly identifies an object's type. For a pfDCS, for instance, **pfGetTypeName()** returns the string "pfDCS".

Chapter 3

“Building a Visual Simulation Application”

This chapter outlines the structure of a simple visual simulation application.

Chapter 3

Building a Visual Simulation Application

This chapter outlines the steps involved in using *libpf*, the visual simulation development library. The outline follows the development sequence of a skeleton application program that introduces you to the basic concepts involved in creating a visual simulation application with *libpf*. Each step at which more complex constructions are possible gives a cross-reference to a later section where you can learn more about the topic.

Overview

It takes only a few lines of code to set up an IRIS Performer *libpf* application. Furthermore, once you have an application framework that you like you can use it again to create other *libpf* applications.

Certain configuration and control routines are required in all applications, while others depend on the features needed and the platform for which the application is designed. The basic requirements for simple programs are the same as for more complex programs, so you can learn the basic structure from a very simple framework application and then build on it to suit your needs.

Take a few moments to browse through the introductory program, *simple.c*, shown in Example 3-1. If you want to compile this program, refer to the section of this chapter titled “Compiling and Linking IRIS Performer Applications.”

Note: Sample code built upon the framework presented in *simple.c* is presented throughout the remainder of this guide, so familiarize yourself with the concepts presented here before moving on to more advanced subjects.

Example 3-1 shows the basic framework of an IRIS Performer application.

Example 3-1 Structure of an IRIS Performer Application

```
#include <stdlib.h>
#include <Performer/pf.h>
#include <Performer/pfutil.h>
#include <Performer/pfdu.h>

int
main (int argc, char *argv[])
{
 float t = 0.0f;
 pfScene *scene;
 pfNode *root;
 pfPipe *p;
 pfPipeWindow *pw;
 pfChannel *chan;
 pfSphere bsphere;

 if (argc < 2)
 {
   pfNotify(PFNFY_FATAL, PFNFY_USAGE,
     “Usage: simple file.ext\n”);
   exit(1);
 }

 /* Initialize Performer */
 pfInit();

 /*
  * Select multiprocessing mode based on
  * number of processors
  */
 pfMultiprocess( PFMP_DEFAULT );

 /* Load all loader DSO’s before pfConfig() forks */
 pfdInitConverter(argv[1]);

 /*
  * Initiate multi-processing mode set by pfMultiprocess
  * FORKs for Performer processes, CULL and DRAW, etc.
  * happen here.
  */
 pfConfig();
```

```
/*
 * Append to Performer search path, PFPATH, files in
 * /usr/share/Performer/data
 */
pfFilePath(“.:/usr/share/Performer/data”);

/* Read a single file, of any known type. */
if ((root = pfdLoadFile(argv[1])) == NULL)
{
pfExit();
exit(-1);
}

/* Attach loaded file to a new pfScene. */
scene = pfNewScene();
pfAddChild(scene, root);

/* Create a pfLightSource and attach it to scene. */
pfAddChild(scene, pfNewLSource());

/* Configure and open graphics window */
p = pfGetPipe(0);
pw = pfNewPWin(p);
pfPWinType(pw, PFPWIN_TYPE_X);
pfPWinName(pw, “IRIS Performer”);
pfPWinOriginSize(pw, 0, 0, 500, 500);

/* Open and configure the GL window. */
pfOpenPWin(pw);

/* Create and configure a pfChannel. */
chan = pfNewChan(p);
pfChanScene(chan, scene);
pfChanFOV(chan, 45.0f, 0.0f);

/* determine extent of scene’s geometry */
pfGetNodeBSphere (root, &bsphere);
pfChanNearFar(chan, 1.0f, 10.0f * bsphere.radius);

/* Simulate for twenty seconds. */
while (t < 20.0f)
{
  pfCoord view;
  float s, c;
```

```
    /* Compute new view position. */
    t = pfGetTime();
    pfSinCos(45.0f*t, &s, &c);
    pfSetVec3(view.hpr, 45.0f*t, -10.0f, 0);
    pfSetVec3(view.xyz, 2.0f * bsphere.radius * s,
      -2.0f * bsphere.radius *c,
       0.5f * bsphere.radius);
    pfChanView(chan, view.xyz, view.hpr);

    /* Initiate cull/draw for this frame. */
    pfFrame();
  }

  /* Terminate parallel processes and exit. */
  pfExit();
}
```

If you would like to compile *simple.c* and try it out, use the copy in */usr/share/Performer/src/pguide/libpf/C*; the *Makefile* in that directory provides all the necessary compilation options. (For more information about IRIS Performer compiler options, see the "Compiling and Linking IRIS Performer Applications" section of this chapter.) Once you've compiled the code, try executing it with some of the sample data files in */usr/share/Performer/data*, such as *blimp.flt* or *sampler.nff*.

Here's a description of the steps involved in a simple IRIS Performer application. Refer to the sample code in Example 3-1 as you read through these steps.

1. Include the necessary system header files.

   ```
   #include <stdlib.h>
   ```

2. Include the relevant IRIS Performer header files.

   ```
   #include <Performer/pf.h>
   #include <Performer/pfutil.h>
   #include <Performer/pfdu.h>
   ```

3. Declare variables for the required elements.

pfScene	a scene graph to be rendered on a channel
pfPipe	a graphics pipeline to perform the rendering
pfChannel	a view to be rendered on the designated pipe

   You can configure IRIS Performer to use multiple scenes, multiple pipes (if your system has them), and multiple channels per pipe. (See "Using Multiple Channels" in Chapter 4.)

4. Initialize IRIS Performer.

   ```
   pfInit();
   ```

   This sets up the shared-memory arena used for multiprocessing, initializes the high-resolution clock, and resets IRIS Performer's state.

5. Configure IRIS Performer.

   ```
   pfConfig();
   ```

   This configures the number of pipes and starts processes based on the selected multiprocessing model. The code in Example 3-1 uses the defaults: a single pipe and a multiprocessing model that is tailored to the number of processors on the machine.

6. Load or create the database.

   ```
   root = pfdLoadFile(argv[1])
   ```

   pfdLoadFile() loads a database from the disk using whichever file importer seems appropriate (based on the three-letter extension at the end of the given filename). There are other ways to set up scenes, too; for instance, you can call a specific importing routine in place of **pfdLoadFile()** if you want to load only databases of a particular format, or you can create geometric objects directly using *libpr* and place them in a database hierarchy. See "Geometry" in Chapter 10 for information on constructing pfGeoSets, and "Scene Graph Hierarchy" in Chapter 6 for information on creating a scene graph.

7. Create a new scene for the channel to draw.

   ```
   scene = pfNewScene();
   ```

8. Add the root of the database that you loaded or created in step 6 to the scene.

   ```
   pfAddChild(scene, root);
   ```

9. Initialize a graphics-rendering pipeline.

   ```
   p = pfGetPipe(0);
   pw = pfNewPWin(p);
   pfPWinType(pw, PFPWIN_TYPE_X);
   pfPWinName(pw, “IRIS Performer”);
   pfPWinOriginSize(pw, 0, 0, 500, 500);

   /* Open and configure the graphics window. */
   pfOpenPWin(pw);
   ```

   This sets up a callback to open a graphics library window, sized and positioned as specified in **OpenPipeWin()**.

10. Specify the frame rate and the synchronization method.

    Because neither a frame rate nor a synchronization method is specified in *simple.c*, the application “free runs” without frame-rate control, which is the default. See “Frame Rate and Synchronization” on page 63 and Chapter 7, “Frame and Load Control,” for more information on controlling frame rates.

11. Create a channel on the specified pipe.

    ```
    chan = pfNewChan(p);
    ```

    A channel is a viewport into a pipe. Because *simple.c* doesn’t configure any screen dimensions for the channel, it renders to the full window of the pipe.

12. Configure the channel: set the viewpoint, field-of-view (FOV), and near and far clipping planes (based on the size of the scene).

    ```
    pfChanScene(chan, scene);
    pfChanFOV(chan, 45.0f, 0.0f);
    pfGetNodeBSphere (root, &bsphere);
    pfChanNearFar(chan, 1.0f, 10.0f * bsphere.radius);
    ```

    When you pass in zero as a field of view—in this case, the vertical FOV—IRIS Performer matches the FOV to the aspect ratio of the channel.

13. Render the scene repeatedly until the specified time is up.
    - Set up the viewpoint for the next frame:

      ```
      pfChanView(chan, view.xyz, view.hpr);
      ```

    - Initiate the next cull / draw cycle to render the frame:

      ```
      pfFrame();
      ```

14. When the time is up, exit IRIS Performer.

    ```
    pfExit();
    ```

The remainder of this chapter discusses portions of the outline in detail. You may find it helpful to continue to refer to *simple.c* while you read the following sections.

Setting Up the Basic Elements

This section describes how to set up the basic requirements of an IRIS Performer *libpf* application.

Using IRIS Performer Header Files

The header files for the IRIS Performer libraries are in the */usr/include/Performer* directory. They include *pf.h* and *pr.h* (header files for *libpf* and *libpr*, respectively), *irisgl.h* and *opengl.h* (to set up declarations for whichever graphics library your application uses), and other header files for use with the other IRIS Performer libraries.

The header files contain useful macros as well as function declarations, including macros for transparently casting a variable from one data type to another. (ANSI C requires that expressions used as function arguments be cast to match function prototypes.) Some routines therefore accept more than one type of argument, with automatic casting between usable types. For example, a routine accepting a pfGroup as an argument can also take a pfSwitch. In the code below, *switch* is automatically cast to a pfGroup* and *geode* is automatically cast to a pfNode* by a macro within *pf.h*:

```
pfGeode *geode;
pfSwitch *switch;
pfAddChild(switch, geode);
```

Initializing and Configuring IRIS Performer

Before you can set up a pipe, you have to set up any areas of shared memory that you intend to use, and you have to determine how many processors to use (and in what configuration).

Initializing Shared Memory

IRIS Performer uses shared memory to share data among the application, the visibility cull traversal, and the draw traversal, all of which can run in parallel on different processors. **pfInit()** sets up the shared memory arena from which *libpf* objects are allocated. The shared memory arena uses either swap space or a file in the directory specified by the environment variable PFTMPDIR. For more information on shared memory arenas, see "Memory Allocation" in Chapter 10.

Initializing Processes

pfConfig() starts up multiple processes, which allow visibility culling and drawing to run in parallel with the application process. The number of processes created depends on the process model (specified by a call to **pfMultiprocess()**), the number of processors, and the number of pipes (one by default; call **pfMultipipe()** to specify more than one pipe). The order of the calls is important—**pfMultiprocess()** and **pfMultipipe()** are effective only if called between **pfInit()** and **pfConfig()**.

The default is a single pipe running with one, two (separate draw process), or three (separate cull and draw processes) processes, depending on the number of processors on the machine. When you run the application from the *root* login account, **pfConfig()** also sets nondegradable priorities for the processes to improve the consistency of the run-time behavior.

For information on controlling multiple pipes, see "Using Pipes" in Chapter 4. For information on multiprocessing, see "Successful Multiprocessing With IRIS Performer" in Chapter 7.

In addition to setting up shared memory, **pfInit()** initializes a high-resolution clock by calling **pfInitClock()**. Depending on the hardware, this may start up a process to service the clock. The clock process consumes few system resources because it sleeps most of the time.

Setting Up a Pipe

A pfPipe variable (also called a *pipe*) represents an IRIS Performer software graphics pipeline. You gain access to a pipe using **pfGetPipe()**; for instance,

```
p = pfGetPipe(0);
```

sets *p* to point to the IRIS Performer graphics pipeline numbered zero.

IRIS Performer maintains its own representation of the global graphics state. Therefore, changes that you make to the graphics state using graphics library (IRIS GL or OpenGL) commands can create inconsistencies. IRIS Performer provides state management routines that let you manipulate both the graphics library state and the IRIS Performer state. When you want to change graphics states, use these routines rather than their graphics library counterparts.

See "Using Pipes" in Chapter 4 for more information on configuring graphics on a pipe and "Graphics State" in Chapter 10 for more information on controlling the graphics state.

Frame Rate and Synchronization

The frame rate is the number of times per second the application intends to draw the scene. The period for a frame must be an integer multiple of the video vertical retrace period, which is typically 1/60th of a second. Thus, with a 60 Hz video rate, possible frame rates are 60 Hz, 30 Hz, 20 Hz, 15 Hz, and so on. *simple.c* doesn't specify a frame rate, so it attempts to free run at the default rate of 60 Hz.

The synchronization mode or *phase* defines how the system behaves if drawing takes more than the requested time. Free-running mode (the default) is useful for applications that don't require a fixed frame rate. **pfSync()** delays the application until the next appropriate frame boundary.

See Chapter 7, "Frame and Load Control," to learn more about frame rates, phase, and synchronization modes.

Setting Up a Channel

A channel is a rendering viewport into a pipe. A pipe can have many channels within it, but by default a channel occupies the full window of a pipe. You can tell the channel to use a portion of the window using **pfChanViewport()**:

```
pfChanViewport(chan, left, right, bottom, top);
```

Channels support the standard viewing concepts such as eyepoint, view direction, field of view, and near and far clipping planes.

For displays using multiple adjacent screens, you can slave channels together to a single viewpoint. You can also use channels to control scene management functions such as the switching of level-of-detail models based on graphics stress and pixel size.

See “Using Channels” in Chapter 4 to learn more about setting up channels.

Creating and Loading a Scene Graph

Databases exist in a variety of formats. IRIS Performer doesn’t define a file format for databases; instead, it supports extensible run-time scene definitions of sufficient generality to support many database formats. Source code for several file importers is included with IRIS Performer; the provided importers are described in Chapter 9, “Importing Databases.”

Creating a Database

You can create a database with any modeler, or write your own modeler using *libpr* routines. If you use a modeler that has its own database format, you can develop a file importer for it by modifying one of the sample importers. See Chapter 9, “Importing Databases,” for more information about import routines.

If you write your own modeler using *libpr* routines, you don’t have to convert the data structures for *libpf* to be able to use them. In this case, you create a database by using a series of calls to construct geometry in pfGeoSets, to define state and texture definitions in pfGeoStates, and to construct a scene graph of pfNodes. The sample programs in Chapter 10,

"libpr Basics," show how to construct simple geometry. The source code for the sample importers demonstrate the construction of more complex scenes.

Setting the Search Path for Database Files

Database files are often scattered about a file system, making file-loading operations tedious. IRIS Performer provides a general mechanism for defining multiple search paths.

When IRIS Performer attempts to open a file, it first tries the name as specified. If that fails, it begins to search for the file using a *search path*, which specifies where to look for data.

You can specify a search path using **pfFilePath(path)** or with the environment variable PFPATH. You can specify any number of directories in this way. The search path consists of a colon (:) separated list of absolute or relative path names of the directories where data might reside. Directories are searched in the order given, beginning with those specified in PFPATH, followed by those specified by **pfFilePath()**.

For example, the following function call tells IRIS Performer to search for data first in the current directory, then in the data directory within the current directory, and then in data directories one and two levels above the current directory:

```
pfFilePath(".:./data:../data:../../data");
```

Calls to **pfFindFile()** with the name of the file you want to locate return the complete pathname of the file if the result of the search is successful.

Simulation Loop

After the pipes and channels are configured and the scene is loaded, the main simulation loop begins and manages scene updates, viewpoint updates, scene intersection inquiries, and image generation.

The loop has two principal control calls: **pfSync()** and **pfFrame()**.

The order of operations is this:

1. Call **pfSync()** to put the process to sleep until the next frame boundary. This step is typically only used when viewpoint information is being updated from a streaming input device such as a head-tracker.
2. Perform latency-critical operations such as setting the viewpoint or reading positional input/output devices.
3. Call **pfFrame()** to initiate the next cull traversal.
4. Perform any time-consuming calculations that are required.
5. Return to step 1.

Time-consuming operations such as intersection inquiries and simulator dynamics computations that are performed in the main simulation loop should go after **pfFrame()** but before **pfSync()**. If these calculations are done after **pfSync()** but before **pfFrame()**, the calculations can delay the start of the cull process and thereby reduce the time available for the cull traversal on multiprocessor systems.

Performance

This chapter doesn't specifically discuss performance tuning (see Chapter 13, "Performance Tuning and Debugging," for detailed information on that topic), but every IRIS Performer-based application should be written with performance in mind. Speed is not something that you can easily build into an application as a last-minute addendum; it's something that you need to consider as you structure your database, as you decide what needs to happen in your main loop, and so on—during the design of your program rather than after debugging it. Once you understand the basics of building a visual simulation application, you should go on to learn how to enhance performance by reading Chapter 13. It might be useful to skim Chapter 3 before reading any further, so that as you read more about the details of building an application with IRIS Performer you'll have performance issues in mind from the start.

Compiling and Linking IRIS Performer Applications

This section describes how to compile and link IRIS Performer applications.

Required Libraries

The following libraries are required when linking an executable:

libpf	IRIS Performer visual simulation development library. Comes in two version: *libpf_ogl* and *libpf_igl*, for OpenGL and IRIS GL respectively.
libpr	IRIS Performer high-performance rendering library. Exists in both OpenGL and IRIS GL versions, and is contained within the corresponding

libpf: *libpf_ogl* and *libpf_igl*, for OpenGL and IRIS GL respectively.

libpfdu	IRIS Performer database library; does file handling, and includes importers for a variety of data formats. Comes in two version: *libpfdu_ogl* and *libpfdu_igl*, for OpenGL and IRIS GL respectively.
libpfutil	IRIS Performer utilities library; includes the window-related functions. Comes in two version: *libpfutil_ogl* and *libpfutil_igl*, for OpenGL and IRIS GL respectively.
libimage	Image library—required by *libpr*.
libGLU	OpenGL utilities library—required by *libpr* with *OpenGL*.
libGL	OpenGL graphics library; either this or the alternative *libgl* (the IRIS GL graphics library) is required by *libpf* and *libpr*.
libXext	
libGLw	OpenGL widget library, for using OpenGL with IRIS IM.
libXm	IRIS IM library; used for “Silicon Graphics look” windows.
libXt	X toolkit intrinsics library; used by IRIS IM.
libXmu	
libX11	X Window System ™ library—required by *libgl* and *libpr*.

libfm	IRIS GL font manager—required by *libpfutil* with IRIS GL.
libm	Math library—required by *libpr*.
libfpe	Floating point exception library—required by *libpr*.
libmalloc	Memory allocation library.
libC	C++ library—required by *libpf*.

For an IRIS GL application using all of the libraries included with IRIS Performer, the link line should include:

```
-lpfdu_igl -lpfui -lpfutil_igl -lpf_igl -limage -lgl -lXmu
-lX11 -lm -lfpe -lfm -lmalloc -lC
```

The corresponding line for an OpenGL application would be:

```
-lpfdu_ogl -lpfui -lpfutil_ogl -lpf_ogl -limage -lGLU -lGL
-lXext -lXmu -lX11 -lm -lfpe -lmalloc -lC
```

Dynamic Shared Objects (DSOs)

The standard libraries for IRIS Performer are distributed as *dynamic shared objects*. Compared with static libraries, DSOs produce smaller applications and allow sharing between multiple executables that are running simultaneously. However, if you build an application using a DSO, that DSO must be present on the target system at run time. The DSOs for IRIS Performer 2.0 are in the performer_eoe subsystem on the IRIS Performer CD-ROM.

Debug and Static Libraries

Advanced

IRIS Performer also ships with the its libraries in different forms that might be useful to developers. The debug versions are primarily intended for bug reporting because they contain more symbol table information than the optimized versions. The static versions are for use when distributing an application to customers who may not have performer_eoe installed. If you want to ensure that your customers will have all the libraries they need, you should use static linking. Debug DSO, static optimized and static debug versions of the libraries can be found in optional subsystems and are installed under the directories */usr/lib/Performer/Debug*, */usr/lib/Performer/Static* and */usr/lib/Performer/StaticDebug*, respectively. The “-L” option to cc, CC or ld can be used to link with the static libraries. Use of

the standard DSO or debug DSO is determined at run time through the environment variable LD_LIBRARY_PATH.

Note: See Chapter 9, "Importing Databases," for information concerning file readers, which are normally accessed as DSOs at run time even when the main IRIS Performer libraries have been statically linked. Also, when linking statically, you should not use the *-no_unresolved*, option since IRIS Performer may reference symbols such as OpenGL extensions which are not installed on your machine.

Using Compiler Flags

Much of the sample code in this guide, many of the sample applications, and most of the database-importing code are written in ANSI C. They should be compiled using the –ansi flag to the C compiler.

Using –cckr instead of –ansi affects IRIS Performer in the following ways:

1. Because –cckr doesn't support floating point constants denoted with the *f* suffix, all constants defined with **#define** are double precision. The promotion of floating point expressions to double precision can decrease performance for some numerically intensive applications.
2. Because –cckr doesn't allow a macro to have the same name as a routine, the type-casting macros in *pf.h* are not available. Thus, when you pass a pointer to a derived type such as pfGroup or pfGeode to a routine that takes a generic type such as a pfNode, that argument must be cast to a pfNode explicitly, as shown in the following example:

```
pfGeode *geode;
pfSwitch *switch;
pfAddChild((pfGroup *)switch, (pfNode *)geode);
```

MIPS-3, MIPS-4, and 64-Bit Compilation

If you are running version 6.2 or later of IRIX, you can compile and execute OpenGL-based IRIS Performer applications in 64-bit mode.

To do this, you need to have installed the optional 64-bit versions of the IRIS Performer libraries. All that is required then is to use the "-64" switch to the compiler. This selects the compilation mode and causes libraries to be searched for in */usr/lib64* instead of */usr/lib*.

The 64-bit version of IRIS Performer is itself created using -mips3, so that you can compile an application using either the MIPS-3 or MIPS-4 instruction set. MIPS-3 executables can run on R4400-based machines such as Onyx and Indigo2 as well as on R8000-based machines such as PowerOnyx and PowerIndigo2. MIPS-4 executables can only be run on R8000-based (and subsequent) machines.

Under IRIX 6.2 and later, if you want to use the extended MIPS-3 or MIPS-4 instruction set in a 32-bit application, install the optional "new 32-bit" (N32) version of IRIS Performer and use the "-n32" option to the compiler. The old 32-bit, new 32-bit and 64-bit versions of IRIS Performer can all be installed at the same time as each is installed in a separate directory, */usr/lib*, */usr/lib32 and /usr/lib64*, respectively.

Note: IRIS GL does not exist in "N32" or 64-bit form; you must use OpenGL.

Using IRIS Performer From C++

IRIS Performer provides C++ bindings for all functions as well as C bindings. Most of this guide does not include code examples in C++; however, all sample programs are provided in the IRIS Performer distribution in both C and C++ versions. The structure of a C++ program is largely identical to that of a C program; for examples of IRIS Performer programs using the C++ API, see the */usr/share/Performer/src/pguide/progs* and *apps* directories for examples of equivalent C and C++ programs.

See Chapter 14, "Programming with C++," for a discussion of the differences between programming using the C and C++ programming interfaces.

Chapter 4

“Setting Up the Display Environment”

This chapter describes how to create a display environment by configuring rendering pipelines, channels, and viewpoints.

Chapter 4

Setting Up the Display Environment

libpf is a visual-database processing and rendering system. The visual database has at its root a pfScene (as described in Chapter 5 and Chapter 6). A pfScene is viewed by a pfChannel, which is rendered to a pfPipeWindow by a pfPipe. This chapter describes how to use pfPipes, pfPipeWindows, and pfChannels.

Using Pipes

This section describes *rendering pipelines* (pfPipes) and their implementation in IRIS Performer. Each rendering pipeline draws into one or more windows (pfPipeWindows) associated with a single Geometry Pipeline. A minimum of one rendering pipeline is required, although it is possible to have more than one.

The Functional Stages of a Pipeline

This rendering pipeline comprises three primary functional stages:

APP	Simulation processing, which includes reading input from control devices, simulating the vehicle dynamics of moving models, updating the visual database, and interacting with other networked simulation stations.
CULL	Traverses the visual database and determines which portions of it are potentially visible (a procedure known as *culling*), selects a *level of detail* for each model, sorts objects and optimizes state management, and generates a display list for the draw function.
DRAW	Traverses the display list and issues graphics library commands to a Geometry Pipeline in order to create an image for subsequent display.

Figure 4-1 shows the process flow for a single-pipe system. The application constructs and modifies the scene definition (a pfScene) associated with a channel. The traversal process associated with that channel's pfPipe then traverses the scene graph, building an IRIS Performer *libpr* display list. As shown in the figure, this display list is used as input to the draw process that performs the actual graphics library actions required to draw the image.

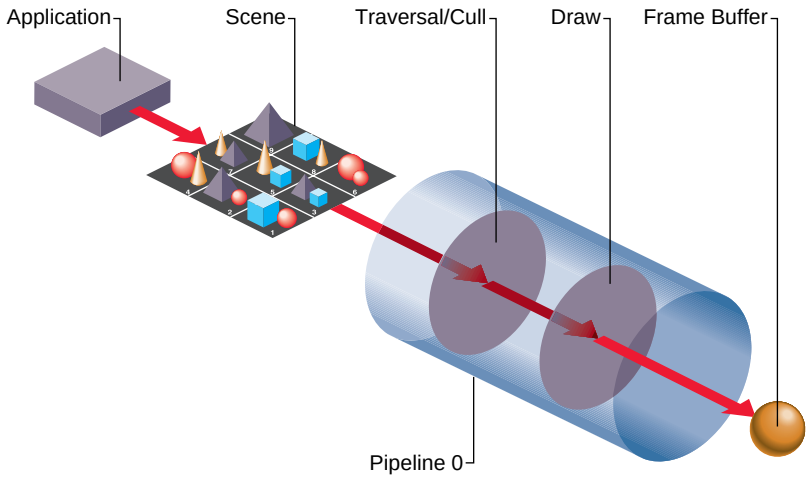


Figure 4-1 Single Graphics Pipeline

IRIS Performer also provides additional processes for application processing tasks, such as database loading and intersection traversals, but these processes are per-application and are asynchronous to the software rendering pipeline(s).

An IRIS Performer application renders images using one or more pfPipes. Each pfPipe represents an independent software-rendering pipeline. Most IRIS systems contain only one Geometry Pipeline, so a single pfPipe is usually appropriate. This single pipeline is often associated with a window that occupies the entire display surface.

Alternative configurations include Onyx systems with RealityEngine^{2}™ graphics (allowing up to three Geometry Pipelines). Applications can render into multiple windows, each of which is connected to a single Geometry Pipeline through a pfPipe rendering pipeline.

Figure 4-2 shows the process flow for a dual-pipe system. Notice both the differences and similarities between these two figures. Each pipeline (pfPipe) is independent in multiple-pipe configurations; the traversal and draw tasks are separate, as are the *libpr* display lists that link them. In contrast, these pfPipes are controlled by the same application process, and in many situations access the same shared scene definition.

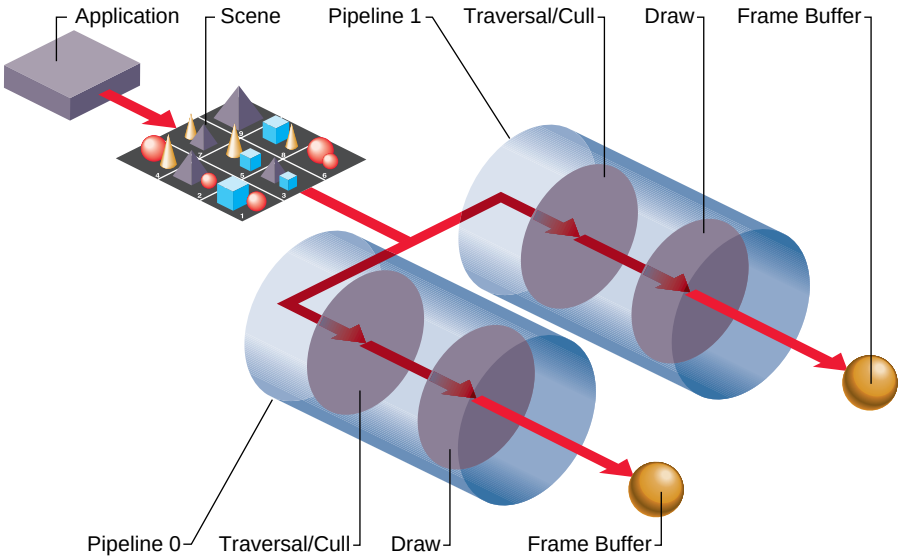


Figure 4-2 Dual Graphics Pipeline

Each of these stages can be combined into a single IRIX process or split into multiple processes (pfMultiprocess) for enhanced performance on multiple CPU systems. Multiprocessing and multiple pipes are advanced topics that are discussed in “Successful Multiprocessing With IRIS Performer” in Chapter 7.

Advanced

Creating and Configuring a pfPipe

pfMultipipe() specifies the number of pfPipes desired. **pfMultiprocess()** specifies the multiprocessing mode used by all pfPipes. These two routines are discussed further in“Successful Multiprocessing With IRIS Performer” in Chapter 7.

pfPipes and their associated processes are created when **pfConfig()** is called. They exist for the duration of the application. After **pfConfig()**, the application can get handles to the created pfPipes using **pfGetPipe()**. The argument to **pfGetPipe()** indicates which pfPipe to return and is an integer between 0 and *numPipes*-1, inclusive. The pfPipe handle is then used for further configuration of the pfPipe.

You may have application state associated with pfPipe stages and processes that need special initialization. For this purpose, you may provide a stage configuration callback for each pfPipe stage using **pfStageConfigFunc(pipe, stageMask, configFunc)** and specify the pfPipe, the stage bitmask (including one or more of PFPROC_APP, PFPROC_CULL, and PFPROC_DRAW), and your stage configuration callback routine. At any time, you may call the function **pfConfigStage()** from the application process to trigger the execution of your stage configuration callback in the process associated with that pfPipe's stage. The stage configuration callback will be invoked at the start of that stage within the current frame (the current frame in the application process, and subsequent frames through the cull and draw phases of the software rendering pipeline). Use a **pfStageConfigFunc()** callback function to configure performer processes not associated with pfPipes, such as the database process, PFPROC_DBASE, and the intersection process, PFPROC_ISECT. A common process initialization task for real-time applications is the selection and / or specification of a CPU on which to run.

The final part of pfPipe initialization is the specification of the graphics hardware pipeline (or screen) and the creation of a window on that screen. The screen of a pfPipe can be set explicitly using **pfPipeScreen()**. Under single pipe operation, pfPipes can also inherit the screen of their first opened window. Under multipipe operation, the screen of all pfPipes must be determined before the pipes are configured by **pfConfigStage()** or the first call to **pfFrame()**. Once the screen of a pfPipe has been set, it cannot be changed. All windows of a pfPipe must be opened on the same screen. A graphics window is associated with a pfPipe through the pfPipeWindow mechanism, which is the subject of the next section, "Using pfPipeWindows." If you do not create a pfPipeWindow, IRIS Performer will automatically create and open a full screen window with a default configuration for your pfPipe.

Once you create and initialize a pfPipe, you can query information about its configuration parameters. **pfGetPipeScreen()** returns the index number of the hardware pipeline for the pfPipe, starting from zero. On single-pipe

systems the return value will be zero. If no screen has been set, the return value will be (-1). **pfGetPipeSize()** returns the full screen size, in pixels, of the rendering area associated with a pfPipe.

Example of pfPipe Use

The sample source code shipped with IRIS Performer includes several simple examples of pfPipe use in both C and C++. Specifically, look at the following examples under the C and C++ directories in */usr/share/Performer/src/pguide/libpf/*, such as *hello.c*, *simple.c*, and *multipipe.c*.

Example 4-1 illustrates the basics of using pipes. The code in this example is adapted from IRIS Performer sample programs.

Example 4-1 pfPipes in Action

```
main()
{
     int i;

     /* Initialize IRIS Performer */
     pfInit();
    /* Set number of pfPipes desired -- THIS MUST BE DONE
     * BEFORE CALLING pfConfig().
     */
    pfMultipipe(NumPipes);
    /* set multiprocessing mode */
    pfMultiprocess(ProcSplit);
    ...
    /* Configure IRIS Performer and fork extra processes if
     * configured for multiprocessing.
     */
    pfConfig();
     ...

    /* Optional custom mapping of pipes to screens.
     * This is actually the reverse as the default.
     *//
     for (i=0; i < NumPipes; i++)
          pfPipeScreen(pfGetPipe(i), NumPipes-(i+1));
    {
        /* set up optional DRAW pipe stage config callback */
        pfStageConfigFunc(-1 /* selects all pipes */,
```

```
                        PFPROC_DRAW /* stage bitmask */,
                        ConfigPipeDraw /* config callback */);
            /* Config func should be done next pfFrame */
            pfConfigStage(i, PFPROC_DRAW);
        }
        InitChannels();
        ...
        /* trigger the configuration and opening of pfPipes
         * and pfWindows
         */
        pfFrame();

        /* Application's simulation loop */
        while(!SimDone())
        {
        ...
        }
}

/* CALLBACK FUNCTIONS FOR PIPE STAGE INITIALIZATION */
void
ConfigPipeDraw(int pipe, uint stage)
{
/* Application state for the draw process can be initialized
 * here. This is also a good place to do real-time
 * configuration for the drawing process, if there is one.
 * There is no graphics state or pfState at this point so no
 * rendering calls or pfApply*() calls can be made.
 */
    pfPipe *p = pfGetPipe(pipe);
    pfNotify(PFNFY_INFO, PFNFY_PRINT,
       "Initializing stage 0x%x of pipe %d", stage, pipe);
}
```

Using pfPipeWindows

IRIS Performer can automatically create and open a full screen window with a default configuration for your pfPipe. At the other extreme, you can create and configure your own windows and set them on a pfPipe. Your window may be pure IRIS GL, IRIS GLX (also known as *mixed model*), or OpenGL / X. In all cases, the pfPipeWindow is the mechanism by which a pfPipe knows about and keeps track of the windows to which it is to render. For all window types, there is a single interface for creating, configuring, and managing the windows.

In the simplest case, IRIS Performer will automatically create a pfPipeWindow for the application and automatically open a full screen window upon the first call to **pfFrame()**. This trivial case is demonstrated in Example 4-1.

Creating, Configuring and Opening pfPipeWindow

In most cases, there are some window parameters, such as size and origin, that you will want to set. You may also have custom graphics state that you need to set to fully initialize your rendering window. This section describes the basics for setting up windows through the pfPipeWindow mechanism. Rendering to the pfPipeWindow is done through a pfChannel's draw process callback and is discussed in the next section in "Creating and Configuring a pfChannel".

A pfPipeWindow can be created for a pfPipe using **pfNewPWin(pipe)**. If you create a pfPipeWindow, then you are responsible for explicitly opening it. The call to **pfOpenPWin(pwin)** from the application process will cause the next call to **pfFrame()** to trigger the opening of the pfPipeWindow in the draw process. A pfPipeWindow created in the application will be a rubber-band window of undefined size for the user to stretch out. This is in contrast to the full screen window that IRIS Performer creates on your behalf in the fully automatic case. To easily get a full screen window, you can use the **pfPWinFullScreen()** function. **pfPWinOriginSize()** can be used to set a fixed position and size for the window. The code in Example 4-2, placed in the application process, will create and open a window in the lower-left corner of the screen of size 500 pixels on each side.

Example 4-2 Creating a pfPipeWindow

```
main()
{
    pfPipe *pipe;
    pfPipeWindow *pwin;
    pfInit();
    ....
    pfConfig();
    /* Create pfPipeWindow for pfPipe 0 */
    pipe = pfGetPipe(0;
    pwin = pfNewPWin(pipe);
    /* Set the origin and size of the pfPipeWindow */
    pfPWinOriginSize(pwin, 0, 0, 500, 500);
    /* Tell IRIS Performer that the pfPipeWindow is ready to
     * be opened
     */
    pfOpenPWin(pwin);
    /* trigger the opening of the pfPipeWindow
     * in the draw process
     */
    pfFrame();
    ...
    while(!SimDone())
    {
    ...
    }
}
```

pfPipeWindows are actually built upon *libpr* pfWindows, but have added support for handling the multiprocessed environment of *libpf* applications and fit into the *libpf* display hierarchy of pfPipes, pfPipeWindows, and pfChannels. Additionally, pfPipeWindows support the multiprocessing environment of *libpf* by having a separate copy of each pfPipeWindow in each pipeline process. All of the “windowness” of pfPipeWindows really comes from the fact that there is a pfWindow internal to the pfPipeWindow. Many of the basic support routines, such **pfPWinFullScreen()** and **pfWinFullScreen()**, have very similar functionality for pfWindows and pfPipeWindows. However, there are situations where pfPipeWindows are able to provide the same functionality in a much more efficient manner. Management of dynamic window origin and size is one case where pfPipeWindows have a real advantage over pfWindows. pfPipeWindows are able to take advantage of the multiprocessed *libpf* environment to always

be able to return an accurate window size and origin relative to the window parent. A process separate from the rendering process is notified by the window system of changes in the pfPipeWindow's size in an efficient manner without impacting the window system or the rendering process. Further details regarding setting and querying window origin and size are discussed with pfWindows in Chapter 10, "libpr Basics."

Note: pfPWin*() routines expect a pfPipeWindow and the pfWin*() routines a pfWindow(). These routines are not interchangeable; pfWindow routines cannot accept pfPipeWindows and visa versa. Some details of pfWindow (and thus pfPipeWindow) functionality are discussed with pfWindows.

Windows have some intrinsic type attributes that must be set before the window is opened. The selection of the screen of a window is determined by the pfPipe that it is opened on, or set for both the pfWindow and its pfPipe with the call **pfPWinScreen()**, or else when the window is finally opened. The window system configuration of the window must also be set before the window is opened. Windows under OpenGL operation will always be X windows. However, under IRIS GL operation, a pfPipeWindow will by default be a pure IRIS GL window. To render IRIS GL into an X window, the X window type must be specified with the command, **pfPWinType(pwin, PFPWIN_TYPE_X)**. An open window must be closed for its type to be changed. The window type argument is actually a bitmask and the type of a pfPipeWindow can include the attributes listed in Table 4-1.

Table 4-1 pfPWinType Tokens

Token	Type Attributes
PFPWIN_TYPE_X	Rendering will be done to an X window. Ignored by OpenGL as all OpenGL rendering is done to X windows.
PFPWIN_TYPE_STATS	The window's normal drawing configuration will support graphics statistics. This affects framebuffer configuration and fill statistics.
PFPWIN_TYPE_SHARE	The pfPipeWindow will automatically be attached to the first pfPipeWindow of the parent pipe with **pfAttachPWin()**

pfPipeWindows have a target default framebuffer configuration. The ability to meet this target will depend on the current graphics hardware configuration, as well as their type. The following parameters are part of the target default configuration and are listed in their order of priority. If the goal framebuffer configuration cannot be created on the current graphics hardware configuration, lower priority parameters will be downgraded as specified.

- double buffered,
- RGB mode with eight bits per color component (four if eight cannot be supported),
- z-buffer with depth of 24 bits,
- one bit stencil buffer (windows type PFWIN_TYPE_STATS will still require 4 bits of stencil),
- multisample buffer of eight, four, or zero samples as available.
- four bit stencil buffer.

pfPipeWindows have IRIS Performer *libpr* rendering state automatically initialized when they are opened. Additionally the following graphics state is automatically initialized upon opening, or upon any call to **pfInitGfx()** for an open window:

- in pure IRIS GL windows, the framebuffer configuration is restored to default; however, if multisample buffers already exist, the default multisampled configuration is used,
- RGB mode is enabled,
- z-buffer is enabled and a z range is set,
- viewport clipping is enabled,
- subpixel vertex accuracy is enabled,
- the viewing matrix is initialized to a two-dimensional one-to--one mapping from eye coordinates to window coordinates.,
- the model matrix is initialized to the identity matrix and made the current GL matrix,
- backface removal is enabled,
- smooth shading is enabled,

- if the current graphics hardware platform supports multisampling, multisampled antialiasing will be enabled with **pfAntialias(PFAA_ON)**,
- a default modulating texture environment is created,
- a default lighting model is created.

Custom framebuffer configuration for a pfPipeWindow can be specified with **pfPWinFBConfigAttrs()**, **pfPWinFBConfig()**, and **pfChoosePWinFBConfig()**. These routines have identical functionality as each of the corresponding pfWindow routines. However, the function **pfChoosePWinFBConfig()** has the constraint that it be called in the draw process because it creates and stores internal data from the window server that must be kept local to the process in which it is called. Table 4-2 lists the different pfPipeWindow routines and describes multiprocessing constraints.

The flexibility in changing the framebuffer configuration of a pfPipeWindow is GL dependent. IRIS GL supports reconfiguration of the framebuffer. However, in GLX or OpenGL/X windows, it is considerably trickier. The main window can remain in place but structures under it must be switched or replaced. If multiple framebuffer configurations are likely to be desired, multiple graphics contexts can be created for the window using pfWindows. pfPipeWindows and pfWindows allow you to have a list of alternate pfWindows that render to exactly the same screen area but may have different framebuffer configuration. You can then select the current configuration for a pfPipeWindow with **pfPWinIndex()**. There are two kinds of common alternate configuration windows that can be created automatically for you: overlay windows created in the overlay planes and windows to support hardware fill statistics (discussed in Chapter 12, “Statistics”). You can use **pfPWinMode()** to indicate that you would like these windows created for you automatically. Special tokens to **pfPWinIndex()** are used to select these common special alternate configuration windows—PFWIN_GFX_WIN, PFWIN_OVERLAY_WIN and PFWIN_STATS_WIN—where PFWIN_GFX_WIN selects the normal default drawing window. Note that only a pfWindow, never a pfPipeWindow, can be an alternate configuration window. Further details on creating and using alternate configuration windows are discussed with pfWindows in Chapter 10, “libpr Basics.” The source code in Example 4-3 is taken from */usr/share/Performer/src/pguide/libpf/C/pipewin.c* and demonstrates the automatic creation and selection of overlay and statistics windows for a pfPipeWindow. This also shows usage of pfChannels and interaction

between pfPipeWindows and pfChannels that will be discussed further in the Section “Creating and Configuring a pfChannel”.

Example 4-3 pfPipeWindow with alternate configuration windows

```
main()
{
    pfPipe *pipe;
    pfPipeWindow *pwin;
    pfInit();
    ....
    pfConfig();

    /* Create pfPipeWindow for pfPipe 0 */
    pipe = pfGetPipe(0);
    pwin = pfNewPWin(pipe);
    /* request automatic default overlay and stats windows */
    pfPWinMode(pwin, PFWIN_HAS_OVERLAY, PF_ON);
    pfPWinMode(pwin, PFWIN_HAS_STATS, PF_ON);
    /* Open the main graphics window */
    pfOpenPWin(pwin);
    pfFrame();

    while(!SimDone())
    {
    ...
    if (Shared->winSel == PFWIN_STATS_WIN))
    {
        /* select statistics window and enable fill stats */
        pfPWinIndex(Shared->pw, PFWIN_STATS_WIN);
        pfFStatsClass(fstats,
              PFSTATSHW_ENGFXPIPE_FILL, PFSTATS_ON);
        pfEnableStatsHw(PFSTATSHW_ENGFXPIPE_FILL);
     }
     else
    {
        /* we are not doing statistics so turn them off */
        pfFStatsClass(fstats,
            PFSTATSHW_ENGFXPIPE_FILL, PFSTATS_OFF);
        pfDisableStatsHw(PFSTATSHW_ENGFXPIPE_FILL);
        pfPWinIndex(Shared->pw, Shared->winSel);
    ...
    }
}
```

```
/* Channel draw process drawing function */
void DrawFunc(void pfChannel *chan)
{
    pfPipeWindow *pwin;
    pwin = pfGetChanPWin(chan);
    if (pfGetPWinIndex(pwin) == PFWIN_OVERLAY_WIN)
    {
        /* Draw overlay image */
        DrawOverlay();
        /* Put back the normal drawing window */
        pfPWinIndex(pwin, PFWIN_GFX_WIN);
        /* Indicate that we will now draw to the window */
        pfSelectPWin(pwin);
     }
     /* call the main IRIS Performer drawing function */
     pfDraw();

}
```

Notice that in Example 4-3, although the pfPipeWindow is doublebuffered, the front and back color buffers are never swapped. This operation is done automatically after all channels on the parent pfPipe have completed their drawing for the given frame.

You may need to set additional window and graphics state to complete the initialization of your pfPipeWindow. Calling **pfOpenPWin()** from the application process does not give you the opportunity to do this. However, with **pfPWinConfigFunc()**, you can supply a window configuration callback function that will enable you to open and initialize your pfPipeWindow in the draw process. A call to **pfConfigPWin()** will trigger one call of the window configuration callback in the draw process upon the next call to **pfFrame()**. **pfConfigPWin()** can be called at any time to trigger the calling of the current window configuration function in the draw process. Example 4-4 demonstrates initializing a pfPipeWindow from a draw process callback. It creates a special extra depth buffer and local light model for supporting IRIS GL shadows (see the */usr/share/Performer/src/pguide/libpf/C/shadow.c* example).

Example 4-4 Custom initialization of pfPipeWindow state

```
main()
{
    pfPipe *pipe;
```

```
    pfPipeWindow *pwin;
    pfInit();
    ....
    pfConfig();

    /* Create pfPipeWindow for pfPipe 0 */
    pipe = pfGetPipe(0);
    pwin = pfNewPWin(pipe);
    /* Set the configuration function for the pfPipeWindow */
    pfPWinConfigFunc(pwin, OpenPipeWindow);
    /* Indicate that OpenPipeWindow should be called in the
     * draw process upon next call to pfFrame().
     */
    pfConfigPWin(pwin);

    /* trigger OpenPipeWindow to be called in the draw
     * process
     */
    pfFrame();
    while(!SimDone())
    {
    ...
    }
}

/* Initialize graphics state in the draw process */
void
OpenPipeWindow(pfPipeWindow *pw)
{
         pfLightModel *lm;

    /* Set some configuration stuff */
    pfPWinOriginSize(pw, 0, 0, 500, 500);
    /* Open the window - will give us initialized libpr and
     * graphics state
     */
    pfOpenPWin(pw);

     /* Shadows require a 32 bit non-multisampled zbuffer */
     do
     {
          zbsize(32);
          stensize(1);
          mssize(numSamples, 32, 1);
          gconfig();
```

```
        numSamples >>= 1;
    } while (PF_ABS(getgconfig(GC_BITS_ZBUFFER)) < 32
                            && numSamples > 0);

    if (numSamples > 0)
            multisample(1);
    else
    {
        RGBsize(12);
        mssize(0, 0, 0);
        zbsize(32);
        stensize(1);
        gconfig();
        multisample(0);
    }
    /* set up Local Light Model to go with shadows */
    lm = pfNewLModel(NULL);
    pfLModelLocal(lm, 1);
    pfApplyLModel(lm);

}
```

Notice how in Example 4-4 the functions **pfPWinOriginSize()** and **pfOpenPWin()** are now called in the draw process, as opposed to the application process as in Example 4-2. In general, configuring or editing any *libpf* object must be done in the application process. pfPipeWindows must be created in the application process. However, pfPipeWindows may be configured, edited, opened and closed in the **pfPWinConfigFunc()** configuration callback which will be called in the draw process. Window operations are best done in a configuration callback, though they can also be done in the drawing callback for a pfChannel on the window. Any function which aspires to directly affect the graphics context must be called in the drawing process. Table 4-2 shows which processes (application or draw via a configuration function) that pfPipeWindow calls can be made from and further detail about these functions can be found in the discussion of pfWindows in Chapter 10, “libpr Basics.”.

Table 4-2 Processes from which to call main pfPipeWindow functions

pfPipeWindow Function	Application Process	Draw process
pfNewPWin()	Yes.	No.
pfPWinMode()	Yes,	Yes.
pfPWinIndex()	Yes,	Yes.
pfPWinConfigFunc()	Yes.	No.
pfOpenPWin() **pfClosePWin()** **pfClosePWinGL()**	Yes.	Yes.
pfPWinOriginSize() **pfPWinFullScreen()**	Yes.	Yes.
pfGetPWinCurOriginSize() **pfGetPWinCurScreenOriginSize()**	X — Yes. IRIS GL — No.	Yes.
pfPWinFBConfigAttrs()	Yes.	Yes.
pfChoosePWinFBConfig()	No.	Yes.
pfPWinFBConfig()	Yes, but the pfFBConfig* must be valid for access in the draw process.	Yes.
pfPWinType() **pfPWinScreen()** **pfPWinShare()**	Yes (before opened).	Yes (before opened).
pfPWinWSWindow() **pfPWinWSDrawable()**	X — Yes. IRIS GL — ID must be valid in the draw process.	Yes,
pfPWinGLCxt()	Yes, but the context must be created in the draw process.	Yes.
pfQueryWin() **pfMQueryWin()**	No.	Yes.

Example 4-3 showed a case where custom IRIS GL code was used in the **pfPWinConfigFunc()** callback to configure the framebuffer for a window. However, IRIS Performer provides GL independent framebuffer configuration utilities. In most cases, **pfPWinFBConfigAttrs(pwin, attrs)** can be used to select a framebuffer configuration for your pfPipeWindow based on the array of attribute tokens, *attrs*. If *attrs* is NULL, the default framebuffer configuration will be selected. If *attrs* is not NULL, the rules for default values follow the rules for configuring windows in OpenGL and X which are different from values in the IRIS Performer default window configuration. Window framebuffer configuration for pfPipeWindows is identical to that of pfWindows and is discussed in more detail in Chapter 10, “libpr Basics,” but the following is a simple example of the specification of framebuffer configuration taken from the sample source code example program */usr/share/Performer/src/pguide/libpf/C/pipewin.c*:

Example 4-5 Configuration of a pfPipeWindow Framebuffer

```
static int FBAttrs[] = {
    PFFB_RGBA,
    PFFB_DOUBLEBUFFER,
    PFFB_DEPTH_SIZE, 24,
    PFFB_RED_SIZE, 8,
    PFFB_SAMPLES, 8,
    PFFB_STENCIL_SIZE, 1,
    NULL,
};

main()
{
    pfPipe *pipe;
    pfPipeWindow *pwin;
    pfInit();
    ....
    pfConfig();

    /* Create pfPipeWindow for pfPipe 0 */
    pipe = pfGetPipe(0);
    pwin = pfNewPWin(pipe);
    /* Set the framebuffer configuration */
    pfPWinFBConfigAttrs(Shared->pw, FBAttrs);
    /* Indicate that the window is ready to open */
    pfOpenPWin(pwin);
    /* trigger the opening of the window in the draw */
```

```
    pfFrame();
    ...
}
```

If you want to do all of your own window creation and management you can do so and just give IRIS Performer the handles to your windows with the **pfPWinWSDrawable()** function; you may also provide a parent X window with the **pfPWinWSWindow()** function. **pfOpenPWin()** will make use of any windows that have already been provided. More details regarding the creation and configuration of pfPipeWindows and pfWindows are discussed in Chapter 10, “libpr Basics.”

pfPipeWindows in Action

pfPipeWindows allow for a reasonable amount of flexibility in the running application. Management of channels in pfPipeWindows is discussed later in this chapter in the Section “Using Multiple Channels“. pfPipeWindows can be re-ordered on their parent pfPipe to control the order that they are drawn in with the command **pfMovePWin(pipe, index, pwin)**. pfPipeWindows can be dynamically opened and closed in the application or draw processes with **pfOpenPWin()** and **pfClosePWin()**. Additionally, **pfConfigPWin()** can be re-issued at any time from the application process to call the current window configuration function to dynamically open, close, and reconfigure pfPipeWindows.

The following example is taken from the distributed source code example file */usr/share/Performer/src/pguide/libpf/C/pipewin.c* and demonstrates the dynamic closing of a window from the application process in the simulation loop and the reuse of **pfConfigPWin()** to reopen the window.

Example 4-6 Opening and Closing a pfPipeWindow

```
main()
{
    ...
    /* main simulation loop */
    while (!Shared->exitFlag)
    {
    /* wait until next frame boundary */
    pfSync();
    pfFrame();
```

```
    /* Set view parameters for next frame */
    UpdateView();
    pfChanView(chan, Shared->view.xyz, Shared->view.hpr);

    /* Close pfPipeWindow */
    if (Shared->closeWin == 1)
    {
        pfClosePWin(Shared->pw);
        ct = pfGetTime();
        Shared->closeWin = 2;
    }
    /* then wait two seconds and reconfig window */
    else if ((Shared->closeWin == 2) &&
                        (pfGetTime() - ct > 2.0f))
    {
        pfConfigPWin(Shared->pw);
        Shared->closeWin = 3;
        pfNotify(PFNFY_NOTICE, PFNFY_PRINT, “OPEN”);
}   }
    ...
}
```

You may want your windows to reside within a larger Motif interface and window hierarchy. IRIS Performer supports this and allows you to run the Motif main loop in a separate process so that you can maintain control of your simulation loop. The Motif interface is created in its own process and Motif event handlers and callbacks will be executed in that process. The Motif callbacks set flags in shared memory to communicate with the main application. Part of this communication is the sharing of X windows between IRIS Performer and Motif. The example program */usr/share/Performer/src/pguide/libpf/C/motif.c* demonstrates the basic elements of this integrated IRIS Performer-Motif hook-up.

Using Channels

This section describes how to use pfChannels. A pfChannel is a view of a graphics scene. A pfChannel is a required element for an IRIS Performer application, because it establishes the visual frame of reference for what is rendered in the drawing process.

Creating and Configuring a pfChannel

When you create a new pfChannel, it is attached to a pfPipe for the duration of the application. The pfPipe renders the pfScene viewed by the pfChannel into a pfPipeWindow that is managed by that pipe. The simplest method uses one channel, one window, and one pipe. You can use multiple channels in a single pfWindow on a pfPipe, thereby allowing channels to share hardware resources. Using multiple channels is an advanced topic that is discussed in the section of this chapter on "Using Multiple Channels." For now, focus your attention on understanding the concepts of setting up and using a single channel.

Use **pfNewChan()** to create a new pfChannel and assign it to a pfPipe. pfChannels are automatically assigned to the first pfPipeWindow of the pfPipe. In the sample program, the following statement creates a new channel and assigns it to pipe *p*.

```
chan = pfNewChan(p);
```

The primary function of a pfChannel is to define the view of a scene. A view is fully characterized by a *viewport*, a *viewing frustum*, and a *viewpoint*. The following sections describe how to set up the scene and view for a pfChannel.

Setting Up a Scene

A pfChannel draws the pfScene set by **pfChanScene()**. A channel can draw only one scene per frame but can change scenes from frame to frame. Other pfChannel attributes such as LOD modifications affect the scene. These attributes are described in "pfLOD Nodes" in Chapter 5.

A pfChannel also renders an environmental model known as pfEarthSky. A pfEarthSky defines the method for clearing the channel viewport before rendering the pfScene and also provides environmental effects, including ground and sky geometry and fog and haze. A pfEarthSky is attached to a pfChannel by **pfChanESky()**.

Setting Up a Viewport

A pfChannel is rendered by a pfPipe into its pfPipeWindow. The screen area that displays a pfChannel's view is determined by the origin and size of the window and the channel viewport specified by pfChanViewport. The channel viewport is relative to the lower left corner of the window and ranges from 0 to 1. By default, a pfChannel viewport covers the entire window.

Suppose that you want to establish a viewport that is one-quarter of the size of the window, located in the lower left corner of the window. Use **pfChanViewport(chan, 0.0, 0.25, 0.0, 0.25)** to set up the one-quarter window viewport for the channel *chan*.

You can then set up other channels to render to the other three-quarters of the window. For example, you can use four channels to create a four-way view for an architectural or CAD application. See "Using Multiple Channels" on page 100 to learn more about multiple channels.

Setting Up a Viewing Frustum

A viewing frustum is a truncated pyramid that defines a viewing volume. Everything outside this volume is clipped, while everything inside is projected onto the viewing plane for display. A frustum is defined by

- field-of-view (FOV) in the horizontal and vertical dimensions
- near and far *clipping planes*

A viewing frustum is created by the intersections of the near and far clipping planes with the top, bottom, left, and right sides of the infinite viewing volume formed by the FOV and aspect ratio settings. The aspect ratio is the ratio of the vertical and horizontal dimensions of the FOV.

Figure 4-3 shows the parameters that define a symmetric viewing frustum. To establish asymmetric frusta refer to the pfChannel(3pf) or pfFrustum(3pf) reference pages for further details.

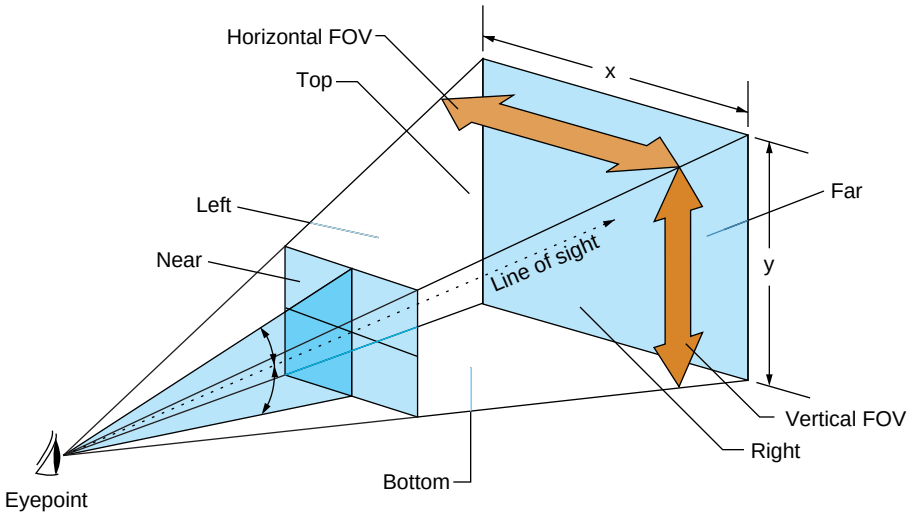


Figure 4-3 Symmetric Viewing Frustum

The viewing frustum is called *symmetric* when the vertical half-angles are equal and the horizontal half-angles are equal.

Field-of-View

The FOV is the angular width of view. Use **pfChanFOV(chan, horiz, vert)** to set up viewing angles in IRIS Performer. The quantities *horiz* and *vert* are the total horizontal and vertical fields of view in degrees; usually you specify one and let IRIS Performer compute the other. If you're specifying one angle, pass any amount less than or equal to zero, or greater than or equal to 180, as the other angle. IRIS Performer automatically computes the unspecified FOV angle to fit the pfChannel viewport using the aspect-ratio preserving relationship

tan(*vert* / 2) / tan(*horiz* / 2) = aspect ratio

That is, the ratio of the tangents of the vertical and horizontal half-angles is equal to the aspect ratio. For example, if *horiz* is 45 degrees and the channel viewport is twice as wide as it is high (so the aspect ratio is 0.5), then the vertical field-of-view angle, *vert*, is computed to be 23.4018 degrees. If both angles are unspecified, **pfChanFOV()** assumes a default value of 45 degrees for *horiz* and computes the value of *vert* as described.

Clipping Planes

Clipping planes define the near and far boundaries of the viewing volume. These distances describe the extent of the visual range in the view, because geometry outside these boundaries is *clipped*, meaning that it isn't drawn.

Use **pfChanNearFar(chan, near, far)** to specify the distance along the line of sight from the viewpoint to the *near* and *far* planes that bound the viewing volume. These clipping planes are perpendicular to the line of sight. For the best visual acuity, choose these distances so that *near* is as far away as possible from the viewpoint and *far* is as close as possible to the viewpoint. Minimizing the range between *near* and *far* provides more resolution for distance comparisons and fog computations.

Setting Up a Viewpoint

A viewpoint describes the position and orientation of the viewer. It is the origin of the viewing location, the direction of the line of sight from the viewer to the scene being viewed, and an up direction. The default viewpoint is at the origin (0, 0, 0) looking along the +Y axis, with +Z up and +X to the right.

Use **pfChanView(chan, point, dir)** to define the viewpoint for the pfChannel identified by *chan*. Specify the view origin for *point* in *x*, *y*, *z* world coordinates. Specify the view direction for *dir* in degrees by giving the degree measures of the three *Euler angles*: *heading*, *pitch*, and *roll*.

Heading is a rotation about the *z* axis, pitch is a rotation about the *x* axis, and roll is a rotation about the *y* axis. The value of *dir* is the product of the rotations ROTy(*roll*) ∗ ROTx(*pitch*) ∗ ROTz(*heading*), where ROT*a*(*angle*) is a rotation matrix about axis *a* of *angle* degrees.

Angles have not only a degree value, but also a *sense*, + or –, indicating whether the direction of rotation is clockwise or counterclockwise. Because different systems follow different conventions, it is very important to understand the sense of the Euler angles as they are defined by IRIS Performer. IRIS Performer follows the *right-hand rule*. According to the right-hand rule, counterclockwise rotations are positive. This means that a rotation about the x axis by +90 degrees shifts the +Y axis to the +Z axis, a rotation about the y axis by +90 degrees shifts the +Z axis to the +X axis, and a rotation about the z axis by +90 degrees shifts the +X axis to the +Y axis.

Figure 4-4 shows a toy plane (somewhat reminiscent of the Ryan S-T) at the origin of a coordinate system with the angles of rotation labeled for heading, pitch, and roll. The arrows show the direction of positive rotation for each angle.

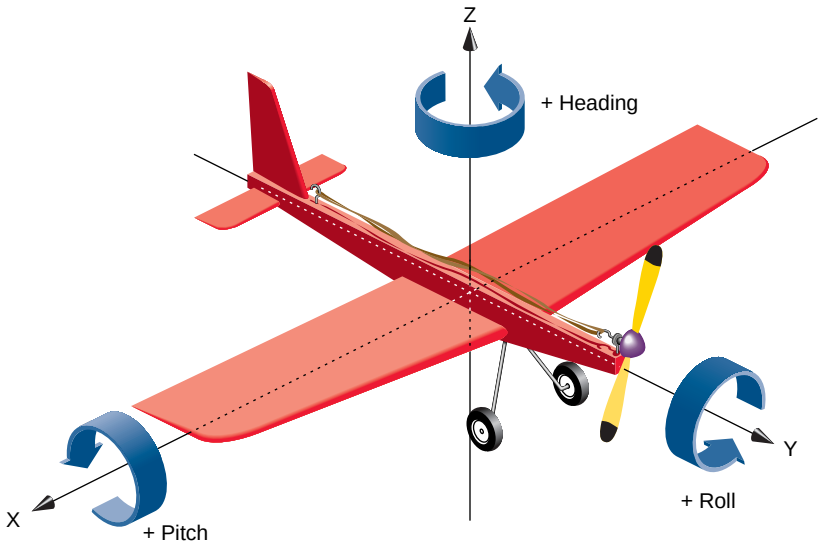


Figure 4-4 Heading, Pitch, and Roll Angles

A roll motion tips the wings from side to side. A pitch motion tips the nose up or down. A yaw motion steers the plane, changing its heading. Accurate readings of these angles are critical information for a pilot during a flight, and a thorough understanding of how the angles function together is required for creation of an accurate flight simulation visual with IRIS Performer. The same is also true of marine and other vehicle simulations.

Alternatively, you can use **pfChanViewMat(chan, mat)** to create a 4x4 homogeneous matrix *mat* that defines the view coordinate system for channel *chan*. The upper left 3x3 submatrix defines the coordinate system axes, and the bottom row vector defines the origin of the coordinate system. The matrix must be orthonormal, or the results will be undefined. You can construct matrices using tools in the *libpr* library.

The origin and heading, pitch, and roll angles, or the view matrix, create a complete view specification. The view specification can locate the eyepoint frame-of-reference origin at any point in world coordinates. The *gaze vector*, the eye's +Y axis, can point in any direction. The *up vector*, the eye's +Z axis, can point in any direction perpendicular to the gaze vector.

You can query the system for the view and eyepoint-direction values with **pfGetChanView(chan, point, dir)**, or obtain the view matrix directly with **pfGetChanViewMat(chan, mat)**.

The view direction can be modified by one or more offsets, relative to the eyepoint frame-of-reference. View offsets are useful in situations where several channels render the same scene into adjacent displays for a wider field-of-view or higher resolution. Offsets are also used for multiple viewer perspectives, such as pilot and copilot views.

Use **pfChanViewOffsets(chan, xyz, hpr)** to specify additional translation and rotation offsets for the viewpoint and direction; *xyz* specifies a translation vector and *hpr* specifies a heading / pitch / roll rotation vector. Viewing offsets are automatically added each frame to the view direction specified by **pfChanView()** or **pfChanViewMat()**.

For example, to create three different perspectives of the same scene as displayed by three windows in an airplane cockpit, use azimuth offsets of 45, 0, and -45 for left, middle, and right views. To create vertical view groups such as might be seen through the windscreen of a helicopter, use both azimuth and elevation offsets. Once the view offsets have been set up, you need only set the view once per frame. View offsets are applied after the eyepoint position and gaze direction have been established. As with the other angles, be aware that the conventions for measuring azimuth and elevation angles vary between graphics systems, so you should verify that the sense of the angles is correct.

Example of Channel Use

Example 4-7 shows how to use various pfChannel-related functions. The code is derived from IRIS Performer sample programs.

Example 4-7 Using pfChannels

```
main()
{
    pfInit();
    ...
    pfConfig();
    ...
    InitScene();
    InitPipe();
    InitChannel();

    /* Application main loop */
    while(!SimDone())
    {
        ...
    }
}

void InitChannel(void)
{
    pfChannel *chan;
    chan = pfNewChan(pfGetPipe(0));

    /* Set the callback routines for the pfChannel */
    pfChanTravFunc(chan, PFTRAV_CULL, CullFunc);
    pfChanTravFunc(chan, PFTRAV_DRAW, DrawFunc);

    /* Attach the visual database to the channel */
    pfChanScene(chan, ViewState->scene);

    /* Attach the EarthSky model to the channel */
    pfChanESky(chan, ViewState->eSky);

    /* Initialize the near and far clipping planes */
    pfChanNearFar(chan, ViewState->near, ViewState->far);

    /* Vertical FOV is matched to window aspect ratio. */
    pfChanFOV(chan, 45.0f/NumChans, -1.0f);
```

```
    /* Initialize the viewing position and direction */
    pfChanView(chan, ViewState->initView.xyz,
               ViewState->initView.hpr);
}

/* CULL PROCESS CALLBACK FOR CHANNEL*/
/* The cull function callback.  Any work that needs to be
 * done in the cull process should happen in this function.
 */
void
CullFunc(pfChannel * chan, void *data)
{
    static long first = 1;

    if (first)
    {
        if ((pfGetMultiprocess() & PFMP_FORK_CULL) &&
            (ViewState->procLock & PFMP_FORK_CULL))
                pfuLockDownCull(pfGetChanPipe(chan));
        first = 0;
    }
    PreCull(chan, data);

    pfCull();               /* Cull to the viewing frustum */

    PostCull(chan, data);
}

/* DRAW PROCESS CALLBACK FOR CHANNEL*/
/* The draw function callback. Any graphics functionality
 * outside IRIS Performer must be done here. I/O with pure
 * IRIS GL devices must happen here.
 */
void
DrawFunc(pfChannel *chan, void *data)
{
    PreDraw(chan, data);     /* Clear the viewport, etc. */

    pfDraw();                /* Render the frame */

    /* draw HUD, read IRIS GL devices, or whatever else needs
     * to be done post-draw.
     */
    PostDraw(chan, data);
}
```

Using Multiple Channels

Each rendering pipeline can render multiple channels to multiple windows. Each channel represents an independent viewpoint into either a shared or an independent visual database. Different types of application can have vastly different pipeline-window-channel configurations. This section describes two basic extremes: visual simulation applications where there is typically one window per pipeline, and highly interactive uses that require dynamic window and channel configuration.

One Window per Pipe, Multiple channels per Window

Often there will is a single channel associated with each pipeline, as shown in the top half of Figure 4-5. This section describes two important uses for multiple-channel support—multiple pipelines per system and multiple windows per pipeline—the second of which is illustrated in the bottom half of Figure 4-5.

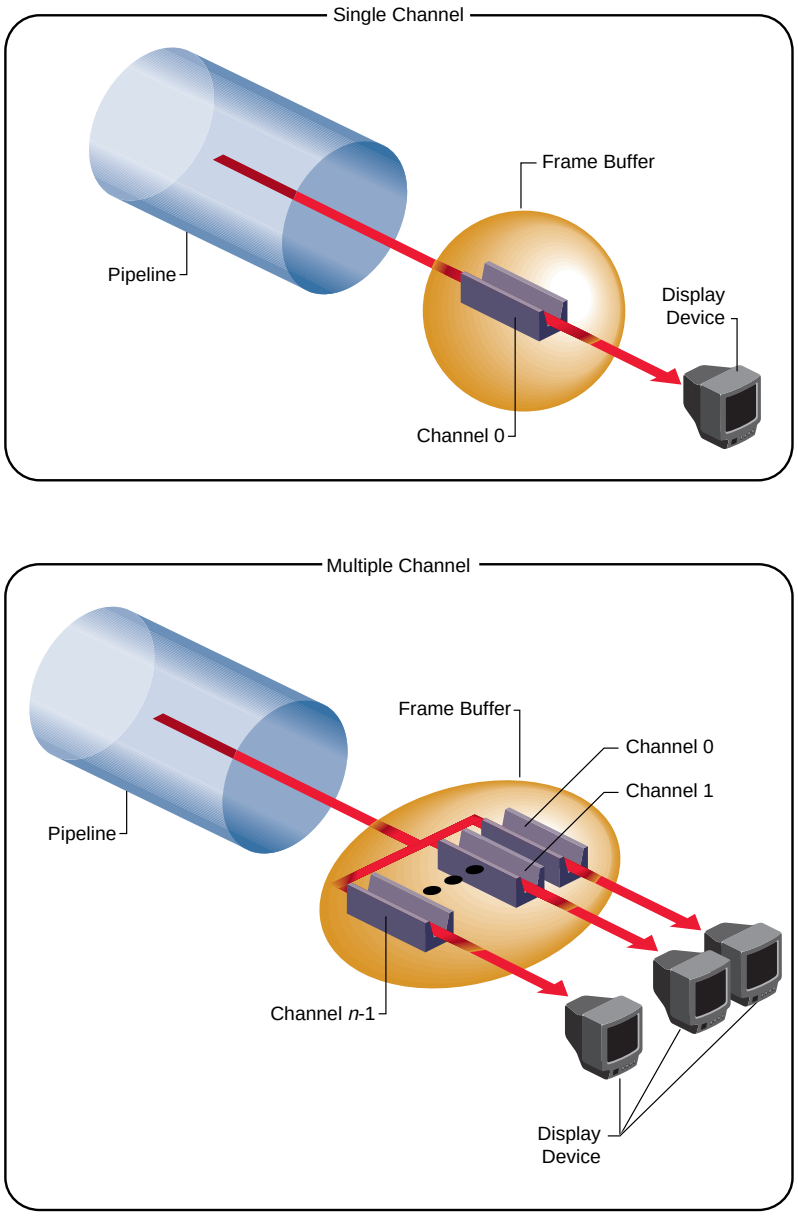


Figure 4-5 Single-Channel and Multiple-Channel Display

One situation that requires multiple channels occurs when inset views must appear within an image. A simple example of this application is a driving simulator in which the screen image represents the view out the windshield. If a rear-view mirror is to be drawn, it must overlay the main forward view to provide a separate view of the same database within the borders of the simulated mirror's frame.

Channels are physically rendered in the order that they are assigned to a pfPipeWindow on their parent pfPipe. Channels, upon creation, are assigned to the end of the channel list of the first window of their pfPipe. In the driving simulator example, creating pipes and channels with the following structure creates two channels on a single shared pipeline:

```
pipeline = pfGetPipe(0);
frontView = pfNewChan(pipeline);
rearView = pfNewChan(pipeline);
```

In this case, IRIS Performer's actual drawing order becomes:

1. Clear *frontView*.
2. Draw *frontView*.
3. Clear *rearView*.
4. Draw *rearView*.

This default ordering results in the rear-view mirror image always overlaying the front-view image, as desired. You can control and re-order the drawing of channels within a pfPipeWindow with the **pfInsertChan(pwin, where, chan)** and **pfMoveChan(pwin, where, chan)** routines. More details about multiple channels and multiple window are discussed in the next section.

When the host has multiple Geometry Pipelines, as supported on Onyx RealityEngine^2 systems, you can create a pfPipe and pfChannel pair for each hardware pipeline. The following code fragment illustrates a two-channel, two-pipeline configuration:

```
leftPipe = pfGetPipe(0);
leftView = pfNewChan(leftPipe);
rightPipe = pfGetPipe(1);
rightView = pfNewChan(rightPipe);
```

This configuration forms the basis for a high-performance stereo display system, since there is a hardware pipeline dedicated to each eye and rendering occurs in parallel.

The two-channel stereo-view application described in this example and the inset-view application described in the previous example can be combined to provide stereo views for a driving simulator with an inset rear-view mirror. The correct management of each eye's viewpoint and the mirror reflection helps provide a convincing sense of physical presence within the vehicle.

The third common multiple-channel situation involves support for multiple video outputs per pipeline. To do this, first associate each pipeline with a set of nonoverlapping channels, one for each desired view. Next, use one of the following video-splitting methods:

- Use the Multi-Channel Option, available from Silicon Graphics for systems such as the Onyx RealityEngine^2, where you can create up to six independent video outputs from a single Graphics Pipeline, with each video output corresponding to one of the tiled channels.
- Connect multiple video monitors in series to a single pipeline's video output. Because each monitor receives the same display image, a masking *bezel* is used to obscure all but the relevant portion of each display surface.

The three multiple-channel concepts described here can be used in combination. For example, use of three RealityEngine^2 pipelines, each equipped with the Multi-Channel Option, allows creation of up to 18 independent video displays. The channel-tiling method can also be used for some or all of these displays.

Example 4-8 shows how to use multiple channels on separate pipes.

Example 4-8 Multiple Channels, One Channel per Pipe

```
pfChannel *Chan[MAX_CHANS];

void InitChannel(int NumChans)
{
    /* Initialize each channel on a separate pipe */
    for (i=0; i< NumChans; i++)
        Chan[i] = pfNewChan(pfGetPipe(i));
```

```
    ...

    /* Make channel n/2 the master channel (can be any
     * channel).
     */
    ViewState->masterChan = Chan[NumChans/2];

    {
        long share;

        /* Get the default channel-sharing mask */
        share = pfGetChanShare(ViewState->masterChan);

        /* Add in the viewport share bit */
        share |= PFCHAN_VIEWPORT;

        if (GangDraw)
        {
            /* add GangDraw to channel share mask */
            share |= PFCHAN_SWAPBUFFERS_HW;
        }
        pfChanShare(ViewState->masterChan, share);
    }

    /* Attach channels */
    for (i=0; i< NumChans; i++)
        if (Chan[i] != ViewState->masterChan)
            pfAttachChan(ViewState->masterChan, Chan[i]);

    ...
    /* Continue with channel initialization */
}
```

Multiple Channels and Multiple Windows

For some interactive applications, you may want to be able to dynamically control the configuration of Channels and Windows. IRIS Performer allows you to dynamically create, open, and close windows, as described in the previous section, “pfPipeWindows in Action“. You can also move channels amongst the windows of the shared parent pfPipe, and re-order channels within a pfPipeWindow. Channels can be appended to the end of a pfPipeWindow channel list with **pfAddChan()** and removed with

pfRemoveChan(). A channel can only be attached to one pfPipeWindow — no instancing of pfChannels is allowed. When a pfChannel is put on a pfPipeWindow, it is automatically deleted from its previous pfPipeWindow. A channel that is not assigned to a pfPipeWindow is not drawn (though it may still be culled).

You can control and re-order the drawing of channels within a pfPipeWindow with the **pfInsertChan(pwin, where, chan)** and **pfMoveChan(pwin, where, chan)** routines. Both of these routines do a type of insertion: **pfInsertChan()** will add *chan* to *pwin*'s channel list in front of the channel in the list at location *where*. **pfMoveChan()** will delete *chan* from it's old location and move it to *where* in *pwin*'s channel list.

Using Channel Groups

In many multiple-channel situations, including the examples described in the previous section, it is useful for channels to share certain attributes. For the three-channel cockpit scenario, each pfChannel shares the same eyepoint while the left and right views are offset using **pfChanViewOffsets()**. IRIS Performer supports the notion of *channel groups*, which facilitate attribute sharing between channels.

pfChannels can be gathered into channel groups that share like attributes. A channel group is created by attaching one pfChannel to another, or to an existing channel group. Use **pfAttachChan()** to create a channel group from two channels or to add a channel to an existing channel group. Use **pfDetachChan()** to remove a pfChannel from a channel group.

A *channel share mask* defines shared attributes for a channel group. The attribute tokens listed in Table 4-3 are bitwise OR-ed to create the share mask.

Table 4-3 Attributes in the Share Mask of a Channel Group

Token	Shared Attributes
PFCHAN_FOV	Horizontal and vertical fields of view
PFCHAN_VIEW	View position and orientation
PFCHAN_VIEW_OFFSETS	(*x*, *y*, *z*) and (*heading*, *pitch*, *roll*) offsets of the view direction
PFCHAN_NEARFAR	Near and far clipping planes
PFCHAN_SCENE	All channels display the same scene
PFCHAN_EARTHSKY	All channels display the same earth/sky model
PFCHAN_STRESS	All channels use the same stress filter
PFCHAN_LOD	All channels use the same LOD modifiers
PFCHAN_SWAPBUFFERS	All channels swap buffers at the same time

Use **pfChanShare()** to set the share mask for a channel group. By default, channels share all attributes except PFCHAN_VIEW_OFFSETS. When you add a pfChannel to a channel group, it inherits the share mask of that group.

A change to any shared attribute is applied to all channels in a group. For example, if you change the viewpoint of a pfChannel that shares PFCHAN_VIEW with its group, all other pfChannels in the group will acquire the same viewpoint.

Two attributes are particularly important to share in adjacent-display multiple-channel simulations: PFCHAN_SWAPBUFFERS and PFCHAN_LOD. PFCHAN_LOD ensures that geometry that straddles displays is drawn the same way in each channel. In this case, all channels will use the same LOD modifier when rendering their scenes so that LOD behavior is consistent across channels. PFCHAN_SWAPBUFFERS ensures that channels refresh the display with a new frame at the same time. pfChannels in different pfPipes that share PFCHAN_SWAPBUFFERS will frame-lock the pipelines together.

Example 4-9 illustrates the use of multiple channels and channel-sharing.

Example 4-9 Channel-Sharing

```
pfChannel *Chan[MAX_CHANS];

main()
{
    pfInit();
    ...
    /* Set number of pfPipes desired.  THIS MUST BE DONE
     * BEFORE CALLING pfConfig().
     */
    pfMultipipe(NumPipes);
    ...
    pfConfig();
    ...
    InitScene();

    InitChannels();

     pfFrame();

    /* Application main loop */
    while(!SimDone())
    {
        ...
    }
}

void InitChannel(int NumChans)
{
    /* Initialize all channels on pipe 0 */
    for (i=0; i< NumChans; i++)
        Chan[i] = pfNewChan(pfGetPipe(0));

    ...

    /* Make channel n/2 the master channel (can be any
     * channel).
     */
    ViewState->masterChan = Chan[NumChans/2];

    ...
```

```
    /* Attach all Channels as slaves to the master channel */
    for (i=0; i< NumChans; i++)
        if (Chan[i] != ViewState->masterChan)
            pfAttachChan(ViewState->masterChan, Chan[i]);

    pfSetVec3(xyz, 0.0f, 0.0f, 0.0f);
    /* Set each channel's viewing offset. In this case use
     * many channels to create one multichannel contiguous
     * frustum with a 45˚ field of fiew.
     */
    for (i=0; i < NumChans; i++)
    {
        float fov = 45.0f/NumChans;

        pfSetVec3(hpr, (((NumChans - 1) * 0.5f) - i) * fov,
                  0.0f, 0.0f);
        pfChanViewOffsets(Chan[i], xyz, hpr);
    }

    ...

    /* Now, just configure the master channel and all of the
     * other channels will share those attributes.
     */

    chan = ViewState->masterChan;
    pfChanTravFunc(chan, PFTRAV_CULL, CullFunc);
    pfChanTravFunc(chan, PFTRAV_DRAW, DrawFunc);
    pfChanScene(chan, ViewState->scene);
    pfChanESky(chan, ViewState->eSky);
    pfChanNearFar(chan, ViewState->near, ViewState->far);
    pfChanFOV(chan, 45.0f/NumChans, -1.0f);
    pfChanView(chan, ViewState->initView.xyz,
               ViewState->initView.hpr);
    ...
}
```

Chapter 5

“Nodes and Node Types”

This chapter describes the structure of IRIS Performer’s scene-definition databases and component data types.

Chapter 5

Nodes and Node Types

An application based on IRIS Performer usually reads in a visual database from a file, then builds an internal data structure incorporating the information in that database. This data structure is called a *scene graph*; each element in the data structure is called a *node*. This chapter describes the types of nodes IRIS Performer uses to represent scenes internally. IRIS Performer provides many functions for creating, querying, modifying, and traversing scene graphs; such functions are also described here.

Note that this chapter focuses on the data types themselves rather than instances of those types. Chapter 6, “Database Traversal,” includes discussion of traversing sample scene graphs, in terms of actual objects rather than abstract data types. Objects in a scene graph (which is to say, instances of node types) have a hierarchical relationship based on the way the scene is laid out. The node types themselves, on the other hand, have an entirely different kind of hierarchical relationship based on the ways in which each node type is like other node types. (For instance, the pfBillboard node type is a subclass of the general pfNode node type.) This concept is discussed in more detail in the section of this chapter titled “Attribute Inheritance” on page 112.

Nodes

A scene is represented by a graph of nodes. *libpf* provides several specialized node types that encapsulate many graphics and visual simulation features. These node types belong to a type hierarchy (which is distinct from the database hierarchy described in Chapter 6). The following sections describe the node types.

Attribute Inheritance

The basic element of a scene hierarchy is the node. While IRIS Performer supplies many specific types of nodes, it also uses a concept called *class inheritance*, which allows different node types to share attributes. For example, many types of nodes have children, but different node types perform different types of operations on their children. Only one set of routines is provided for adding and removing children, but all the node types can use these routines in the same manner to perform these common operations. This reduces the number of routines required and imposes a logical structure on the node types. This logical structure is also known as a *class hierarchy*, but it should not be confused with the scene hierarchy of nodes—the derivation hierarchy contains data types, not objects.

IRIS Performer's node hierarchy begins with the pfNode class, just as the overall class hierarchy begins with the pfObject type (see "Nodes" on page 112). All node types are subordinate to pfNode, and they inherit pfNode's attributes and the *libpf* routines for setting and getting attributes. In general, a node type inherits the attributes and routines of all its parent nodes in the type hierarchy.

Figure 5-1 shows the type hierarchy for the *libpf* node types.

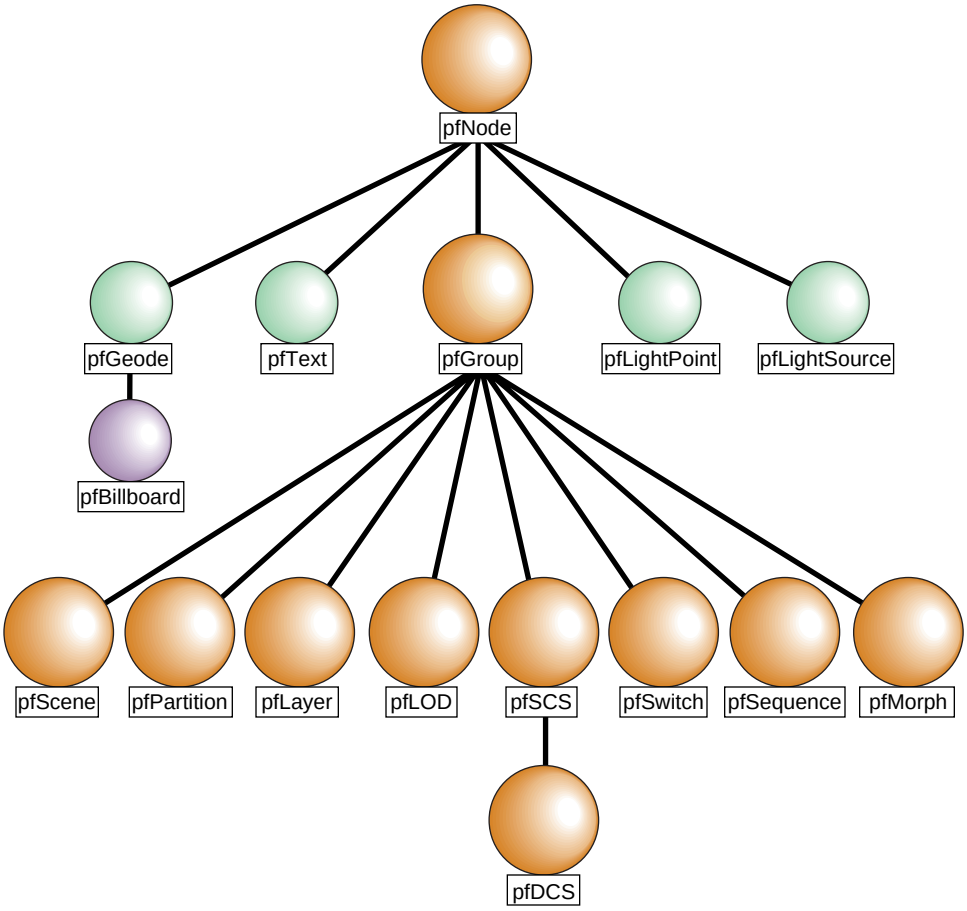


Figure 5-1 Nodes in the IRIS Performer Hierarchy

Table 5-1 lists the basic node class and gives a simple description for each node type.

Table 5-1 IRIS Performer Node Types

Node Type	Node Class	Description
pfNode	Abstract	Basic node type
pfGroup	Branch	Groups zero or more children
pfScene	Root	Parent of the visual database
pfSCS	Branch	Static Coordinate System
pfDCS	Branch	Dynamic Coordinate System
pfSwitch	Branch	Selects among multiple children
pfSequence	Branch	Sequences through its children
pfLOD	Branch	Level-of-detail node
pfLayer	Branch	Renders coplanar geometry
pfLightPoint	Leaf	Contains light point specifications
pfLightSource	Leaf	Contains specifications for a light source
pfGeode	Leaf	Contains geometric specifications
pfBillboard	Leaf	Rotates geometry to face the eyepoint
pfPartition	Branch	Partitions geometry for efficient intersections
pfMorph	Branch	Morphs attribute data of its children
pfText	Leaf	Renders 2D and 3D text

pfNode

As discussed in “Attribute Inheritance” on page 112, all *libpf* nodes are arranged in a type hierarchy, which defines the inheritance of capabilities. The base node class is pfNode; all other nodes inherit the attributes of a pfNode. A pfNode is an abstract class, meaning that a pfNode can never be explicitly created by the application. Its purpose is to provide a root to the type hierarchy and to define the attributes that are common to all node types.

pfNode Attributes

The following pfNode attributes are inherited by all other *libpf* node types:

- Node name
- Parent list
- Bounding geometry
- Intersection and traversal masks
- Callback functions and data
- User data

Bounding geometry, intersection masks, user data, and callbacks are advanced topics that are discussed in Chapter 6, “Database Traversal.”

The routines that set, get, and otherwise manipulate these attributes can be used by all *libpf* node types, as indicated by the keyword ‘Node’ in the routine names. Nodes used as arguments to pfNode routines must be cast to pfNode* to match parameter prototypes, as shown in this example:

```
pfNodeName((pfNode*) dcs, "rotor_rotation");
```

However, you usually don’t need to do this casting explicitly. When you use the C API and compile with the –ansi flag (which is the usual way to compile IRIS Performer applications), *libpf* provides macro wrappers around pfNode routines that automatically perform argument casting for you. When you use the C++ API, such type casting is not necessary.

pfNode Operations

In addition to sharing attributes, certain basic operations are provided for all node types. They include:

New	Create and return a handle to a new node.
Get	Get node attributes.
Set	Set node attributes.
Find	Find a node based on its name.
Print	Print node data.
Copy	Copy node data.
Delete	Delete a node.

The Set operation is implied in the node attribute name. The names of the attribute-getting functions contain the string “Get”.

An Example of Scene Creation

Example 5-1 illustrates the creation of a scene that includes two different kinds of pfNodes. (For information about pfScene nodes, see “pfScene Nodes” on page 125; for information about pfDCS nodes, see “pfDCS Nodes” on page 126.)

Example 5-1 Making a Scene

```
pfScene *scene;
pfDCS *dcs1, *dcs2;

scene = pfNewScene();         /* Create a new scene node */
dcs1 = pfNewDCS();            /* Create a new DCS node */
dcs2 = pfNewDCS();            /* Create a new DCS node */
pfCopy(dcs2, dcs1);           /* Copy all node attributes */
                              /*        from dcs1 to dcs2 */
pfNodeName(scene, "Scene_Graph_Root"); /* Name scene node */
pfNodeName(dcs1,"DCS_1");     /* Name dcs1 */
pfNodeName(dcs2,"DCS_2");     /* Name dcs2 */
...
/* Use a pfGet*() routine to determine node name */
printf("Name of first DCS node is %s.", pfGetNodeName(dcs1));
...
```

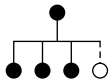
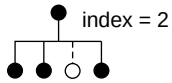
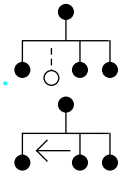
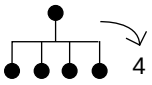
```
/* Recursively free this node if it's no longer referenced */
pfDelete(scene);
...
```

pfGroup

In addition to inheriting the pfNode attributes described in the "pfNode" section of this chapter, a pfGroup also maintains a list of zero or more child nodes that are accessed and manipulated using group operators. Children of a pfGroup can be either branch or leaf nodes. Traversals process the children of a pfGroup in left-to-right order.

Table 5-2 lists the pfGroup functions, with a description and a visual interpretation of each.

Table 5-2 pfGroup Functions

Function Name	Description	Diagram
pfAddChild(group, child)	Appends *child* to the list for *group*.	
pfInsertChild(group, index, child)	Inserts *child* before the child whose place in the list is *index*.	
pfRemoveChild(group, child)	Detaches *child* from the list and shifts the list to fill the vacant spot. Returns 1 if *child* was removed. Returns 0 if *child* was not found in the list. Note that the "removed" node is only detached, not deleted.	
pfGetNumChildren(group)	Returns the number of children in *group*.	

pfGroup nodes can organize a database hierarchy either logically or spatially. For example, if your database contains a model of a town, a logical organization might be to group all house models under a single pfGroup. However, this kind of organization is less efficient than a spatial organization, which arranges geometry by location. A spatial organization improves culling and intersection performance; in the example of the town, spatial organization would consist of grouping houses with their local terrain geometry instead of with each other. Chapter 6 describes how to spatially organize your database for best performance.

The code fragment in Example 5-2 illustrates building a hierarchy using pfGroup nodes.

Example 5-2 Hierarchy Construction Using Group Nodes

```
scene = pfNewScene();

/* The following loop constructs a sample hierarchy by
 * adding children to several different types of group
 * nodes.  Notice that in this case the terrain was broken
 * up spatially into a 4x4 grid, and a switch node is used
 * to cause only one vehicle per terrain node to be
 * traversed.
 */

for(j = 0; j < 4; j++)
    for(i = 0; i < 4; i++)
    {
        pfGroup *spatial_terrain_block = pfNewGroup();
        pfSCS *house_offset = pfNewSCS();
        pfSCS *terrain_block_offset = pfNewSCS();
        pfDCS *car_position = pfNewDCS();
        pfDCS *tank_position = pfNewDCS();
        pfDCS *heli_position = pfNewDCS();
        pfSwitch *current_vehicle_type;
        pfGeode *heli, *car, *tank;

        pfAddChild(scene, spatial_terrain_block);
        pfAddChild(spatial_terrain_block,
                   terrain_block_offset);
        pfAddChild(spatial_terrain_block, house_offset);
        pfAddChild(spatial_terrain_block,
                   current_vehicle_type);
        pfAddChild(current_vehicle_type, car_position);
```

```
        pfAddChild(current_vehicle_type, tank_position);
        pfAddChild(current_vehicle_type, heli_position);
        pfAddChild(car_position, car);
        pfAddChild(tank_position, tank);
        pfAddChild(heli_position, heli);
    }

...

/* The following shows how one might use IRIS Performer to
 * manipulate the scene graph at run time by adding and
 * removing children from branch nodes in the scene graph.
 */

for(j = 0; j < 4; j++)
    for(i = 0; i < 4; i++)
    {
        pfGroup *this_terrain;
        this_terrain = pfGetChild(scene, j*4 + i);
        if (pfGetNumChildren(this_terrain) > 2)
            this_tank = pfGetChild(this_terrain, 2);
        if (is_tank_disable(this_tank))
        {
            pfRemoveChild(this_terrain, this_tank);
            pfAddChild(disabled_tanks, this_tank);
        }
    }
...
```

Working With Nodes

This section describes the basic concepts involved in working with nodes. It explains how shared instancing can be used to create multiple copies of an object, and how changes made to a parent node propagate down to its children. A sample program that illustrates these concepts is presented at the end of the chapter.

Instancing

A scene graph is typically constructed at application initialization time by creating and adding new nodes to the graph. If a node is added to two or more parents it is termed *instanced* and is shared by all its parents. Instancing is a powerful mechanism that saves memory and makes modeling easier. *libpf* supports two kinds of instancing, which are described in the following sections.

Shared Instancing

Shared instancing is the result of simply adding a node to multiple parents. If an instanced node has children, then the entire subgraph rooted by the node is considered to be instanced. Each parent shares the node; thus, modifications to the instanced node or its subgraph are experienced by all parents. Shared instances can be nested—that is, an instance can itself instance other nodes.

In the following sample code, group0 and group1 share a node:

```
pfAddChild(group0, node);
pfAddChild(group1, node);
```

Figure 5-2 shows the structure created by this code. Before the instancing operation, the two groups and the node to be shared all exist independently, as shown in the left portion of the figure. After the two function calls shown above, the two groups both reference the same shared hierarchy. (If the original groups referenced other nodes, those nodes would remain unchanged.) Note that each of the group nodes considers the shared hierarchy to be its own child.

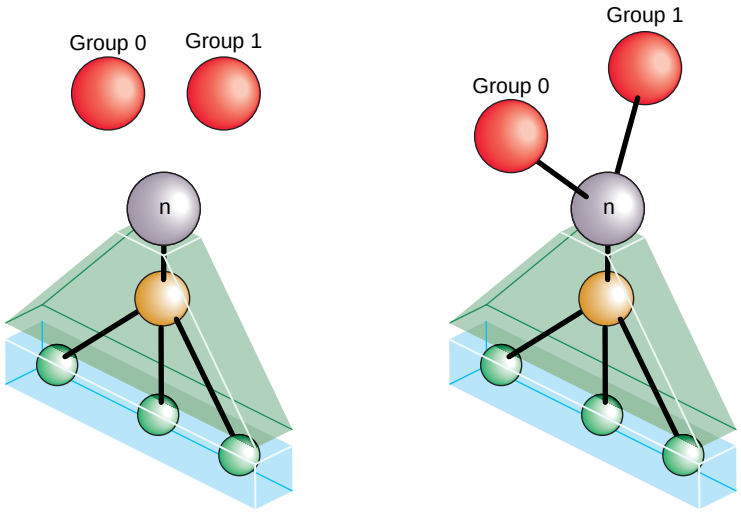


Figure 5-2 Shared Instances

Cloned Instancing

In many situations shared instancing isn't desirable. Consider a subgraph that represents a model of an airplane with articulations for ailerons, elevator, rudder, and landing gear. Shared instances of the model result in multiple planes that share the same articulations. Consequently, it's impossible for one plane to be flying with its landing gear retracted while another is on a runway with its landing gear down.

Cloned instancing provides the solution to this problem by *cloning*—creating new copies of variable nodes in the subgraph. Leaf nodes containing geometry are not cloned and are shared to save memory. Cloning the airplane model generates new articulation nodes, which can be modified independently of any other cloned instance. The cloning operation, **pfClone()**, is actually a traversal and is described in detail in Chapter 6.

Figure 5-3 shows the result of cloned instancing. As in the previous figure, the left half of the drawing represents the situation before the operation, and the right half shows the result of the operation.

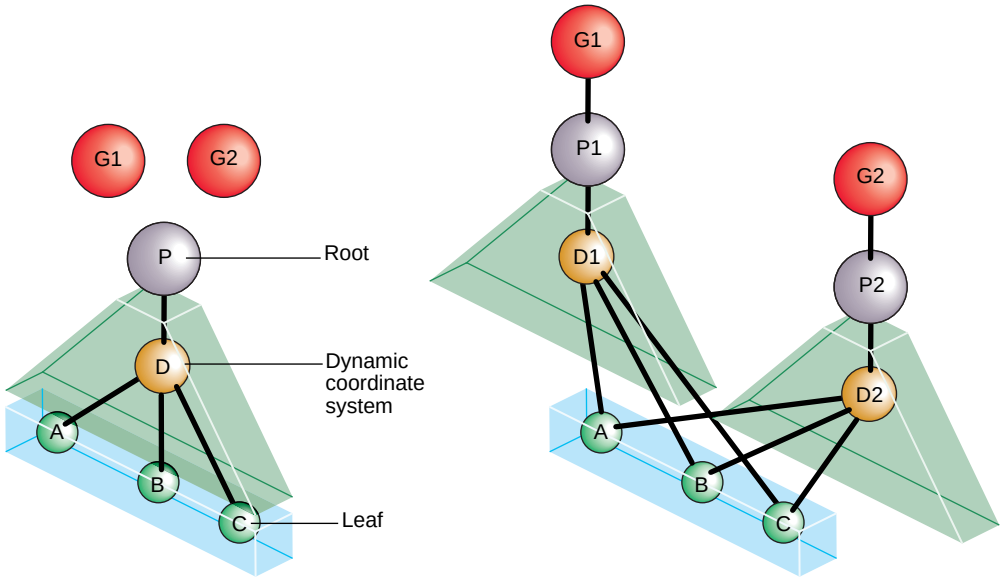


Figure 5-3 Cloned Instancing

The cloned instancing operation constructs new copies of each internal node of the shared hierarchy, but uses the same shared instance of all the leaf nodes. In use, this is an important distinction, because the number of internal nodes may be relatively few, while the number and content of geometry-containing leaf nodes is often quite extensive.

Nodes G1 and G2 in Figure 5-3 are the groups that form the root nodes after the cloned instancing operation is complete. Node P is the parent or root node of the instanced object, and D is a dynamic coordinate system contained within it. Nodes A, B, and C are the leaf geometry nodes; they are shared rather than copied.

The code in Example 5-3 shows how to create cloned instances.

Example 5-3 Creating Cloned Instances

```
pfGroup *g1, *g2, *p;
pfDCS *d;
pfGeode *a, *b, *c;

...
/* Create initial instance of scene hierarchy of p under
 * group g1: add a DCS to p, then add three pfGeode nodes
 * under the DCS.
 */
pfAddChild(g1,p);
pfAddChild(p,d);
pfAddChild(d,a);
pfAddChild(d,b);
pfAddChild(d,c);

...
/* Create cloned instance version of p under g2 */
pfAddChild(g2, pfClone(p,0));
/* Notice that pfGeodes are cloned by instancing rather than
 * copying.  Also notice that the second argument to
 * pfClone() is 0; that argument is currently required by
 * IRIS Performer to be zero.
 */
...
```

Bounding Volumes

libpf uses bounding volumes for culling and to improve intersection performance. *libpf* computes bounding volumes for all nodes in a database hierarchy unless the bound is explicitly set by the application. The bounding volume of a branch node encompasses the spatial extent of all its children. *libpf* automatically recomputes bounds when children are modified.

By default, bounding volumes are *dynamic*; that is, *libpf* automatically recomputes them when children are modified. For instance, in Example 5-4 when the DCS is rotated nothing more needs to be done to update the bounding volume for *g1*.

Example 5-4 Automatically Updating a Bounding Volume

```
pfAddChild(g1,dcs);
pfAddChild(dcs, helicopter);

...

pfDCSRot(dcs, heading+10.0f, pitch,roll);
...
pfDCSRot(dcs, heading, pitch - 5.0f, roll + 2.0f);
```

In some cases, you may not want bounding volumes to be recomputed automatically. For example, in a merry-go-round with horses moving up and down, you know that the horses stay within a certain volume. Using **pfNodeBSphere()**, you can specify a bounding sphere within which the horse always remains and tell IRIS Performer that the bounding volume is "static"—not to be updated no matter what happens to the node's children. You can always force an update by setting the bounding volume to NULL with **pfNodeBSphere()**, as follows:

```
pfNodeBSphere(node, NULL, NULL, PFBOUND_STATIC);
```

At the lowest level, within pfGeoSets, bounding volumes are maintained as axially-aligned boxes. When you add a pfGeoSet to a pfGeode or directly invoke **pfGetGSetBBox()** on the pfGeoSet, a bounding box is created for the pfGeoSet. Neither the bounding box of the pfGeoSet nor the bounding volume of the pfGeode is updated if the geometry changes inside the pfGeoSet. You can force an update by setting the pfGeoSet bounding box and then the pfGeode bounding volume to a NULL bounding box, as follows:

- Recompute the pfGeoSet bounding box from the internal geometry:

  ```
  pfGSetBBox(gset, NULL);
  ```

- Recompute the pfGeode bounding volume from the bounding boxes of its pfGeoSets:

  ```
  pfNodeBSphere(geode, NULL, PFBOUND_DYNAMIC);
  ```

Node Types

This section describes the node types and the functions for working with each node type.

pfScene Nodes

A pfScene is a root node that is the parent of a visual database. Use **pfNewScene()** to create a new scene node. Before the scene can be drawn, you must call **pfChanScene(channel, scene)** to attach it to a pfChannel.

Any nodes that are within the graph that is parented by a pfScene are culled and drawn once the pfScene is attached to a pfChannel. Because pfScene is a group, it uses pfGroup routines; however, a pfScene cannot be the child of any other node. The following statement adds a pfGroup to a scene:

```
pfAddChild(scene, root);
```

In the simplest case, the pfScene is the only node you need to add. Once you have a pfPipe, pfChannel, and pfScene, you have all the necessary elements for generating graphics using IRIS Performer.

pfScene Default Rendering State

pfScene nodes may specify a global pfGeoState that all other pfGeoStates in nodes below the pfScene will inherit from. Specification of this scene pfGeoState is done via the function **pfSceneGState()**. This functionality allows for the subtle optimization of pushing the most frequently used pfGeoState attributes for a particular scene graph into a global state and having the individual states inherit these attributes rather than specify them. This can save Performer work during culling (by having to ‘unwrap’ fewer pfGeoStates) and thus possibly increase frame rate. There are several database utility functions in *libpfdu* designed to help with this optimization. **pfdMakeSceneGState()** returns an ‘optimal’ pfGeoState based on a list of pfGeoStates. **pfdOptimizeGStateList()** takes an existing global pfGeoState, a new global pfGeoState, and a list of pfGeoState’s that should be optimized and cause all attributes of pfGeoStates in the list of pfGeoStates to be inherited if they are the same as the attribute in the new global pfGeoState. Lastly **pfdMakeSharedScene()** will cause this optimization to happen for all of the pfGeoStates under the pfScene that was passed into the function. For

more information on pfGeoStates see Chapter 10, “libpr Basics,” which discusses *libpr* in more detail. For more information of the creation and optimization of databases see Chapter 9, “Importing Databases,” which discusses building database converters and *libpfdu*.

pfSCS Nodes

A pfSCS is a branch node that represents a static coordinate system. A pfSCS node contains a fixed modeling transformation matrix that cannot be changed once it is created. pfSCS nodes are useful for positioning models within a database. For example, a house that is modeled at the origin should be placed in the world with a pfSCS because houses rarely move during program execution.

Use **pfNewSCS(matrix)** to create a new pfSCS using the transformation defined by *matrix*. To find out what matrix was used to create a given pfSCS, call **pfGetSCSMat()**.

For best graphics performance, matrices passed to pfSCS nodes (and the pfDCS node type described in the next section) should be orthonormal (translations, rotations, and uniform scales). Nonuniform scaling requires renormalization of normals in the graphics pipe. Projections and other non-affine transformations are not supported.

While pfSCS nodes are useful in modeling, using too many of them can reduce culling, rendering, and intersection performance. For this reason, *libpf* provides the **pfFlatten()** traversal. **pfFlatten()** will traverse a scene graph and apply static transformations directly to geometry to eliminate the overhead associated with managing the transformations. **pfFlatten()** is described in detail in Chapter 6, “Database Traversal.”

pfDCS Nodes

A pfDCS is a branch node that represents a dynamic coordinate system. Use a pfDCS when you want to apply an initial transformation to a node and also change the transformation during the application. Use a pfDCS to articulate moving parts and to show object motion.

Use **pfNewDCS()** to create a new pfDCS. The initial transformation of a pfDCS is the *identity matrix*. Subsequent transformations are set by specifying a new transformation matrix, or by replacing the rotation, scale, or translation in the current transformation matrix. The pfDCS transforms each child C(i) to C(i)∗Scale∗Rotation∗Translation.

Table 5-3 lists functions for manipulating a pfDCS, including rotating, scaling, and translating the children of the pfDCS.

Table 5-3 DCS Transformations

Function Name	Description
pfNewDCS	Create a new pfDCS node.
pfDCSTrans	Set the translation coordinates to *x*, *y*, *z*.
pfDCSRot	Set the rotation transformation to *h*, *p*, *r*.
pfDCSCoord	Rotate and translate by *coord*.
pfDCSScale	Scale by a uniform scale factor.
pfDCSMat	Use a matrix for transformations.
pfGetDCSMat	Retrieve the current matrix for a given pfDCS.

pfSwitch Nodes

A pfSwitch is a branch node that selects one, all, or none of its children. Use **pfNewSwitch()** to return a handle to a new pfSwitch. To select all the children, use the PFSWITCH_ON argument to **pfSwitchVal()**. Deselect all the children (turning the switch off) using PFSWITCH_OFF. To select a single child, give the index of the child from the child list. To find out the current value of a given switch, call **pfGetSwitchVal()**. Example 5-5 (in the “pfSequence Nodes” section) illustrates a use of pfSwitch nodes to control pfSequence nodes.

pfSequence Nodes

A pfSequence is a pfGroup that sequences through a range of its children, drawing each child for a specified duration. Each child in a sequence can be thought of as a frame in an animation. A sequence can consist of any number of children, and each child has its own duration. You can control whether an entire sequence repeats from start to end, repeats from end to start, or terminates.

Use **pfNewSeq()** to create and return a handle to a new pfSequence. Once the pfSequence has been created, use the group function **pfAddChild()** to add the children that you want to animate.

Table 5-4 describes the functions for working with pfSequences.

Table 5-4 pfSequence Functions

Function	Description
pfNewSeq	Create a new pfSequence node.
pfSeqTime	Set the length of time to display a frame.
pfGetSeqTime	Find out the time allotted for a given frame.
pfSeqInterval	Set the range of frames and sequence type.
pfGetSeqInterval	Find out interval parameters.
pfSeqDuration	Control the speed and number of repetitions of the entire sequence.
pfGetSeqDuration	Retrieve speed and repetition information for the sequence.
pfSeqMode	Start, stop, pause, and resume the sequence.
pfGetSeqMode	Find out the sequence's current mode.
pfGetSeqFrame	Get the current frame.

Example 5-5 demonstrates a possible use of both switches and sequences. First, sequences are set up to contain animation sequences for explosions, fire, and smoke; then a switch is used to control which sequences are currently active.

Example 5-5 Using pfSwitch and pfSequence Nodes

```
pfSwitch *s;
pfSequence *explosion1_seq, *explosion2_seq, *fire_seq,
           *smoke_seq;
...
s = pfNewSwitch();
explosion1_seq = pfNewSeq();
explosion2_seq = pfNewSeq();
fire_seq = pfNewSeq();
smoke_seq = pfNewSeq();

pfAddChild(s, explosion1_seq);
pfAddChild(s, explosion2_seq);
pfAddChild(s, fire_seq);
pfAddChild(s, smoke_seq);
pfSwitchVal(s, PFSWITCH_OFF);
...
if (direct_hit)
{
    pfSwitchVal(s, PFSWITCH_ON); /* Select all sequences */

    /* Set first explosion sequence to go double normal
     * speed and repeat 3 times. */
    pfSeqMode(explosion1_seq, PFSEQ_START);
    pfSeqDuration(explosion1_seq, 2.0f, 3);

    /* Set second explosion sequence to display first child
     * of sequence for 2 seconds before continuing. */
    pfSeqMode(explosion2_seq, PFSEQ_START);
    pfSeqTime(explosion2, 0.0f, 2.0f);

    /* Set fire to wait on first frame of sequence until .3
     * seconds after second explosion. */
    pfSeqMode(fire_seq, PFSEQ_START);
    pfSeqTime(fire_seq, 0.0f, 2.3f);

    /* Set smoke to wait until .1 seconds after fire. */
    pfSeqMode(smoke_seq, PFSEQ_START);
    pfSeqTime(smoke_seq, 0.0f, 2.4f);
}
else if (explosion && (expl_type == 0))
{
    pfSeqMode(explosion1_seq, PFSEQ_START);
    pfSwitchVal(s, 0);
```

```
}
else if (explosion && (expl_type == 1))
{
    pfSeqMode(explosion2_seq, PFSEQ_START);
    pfSwitchVal(s, 1);
}
else if (fire_is_burning)
{
    pfSeqMode(fire_seq, PFSEQ_START);
    pfSwitchVal(s, 2);
}
else if (smoking)
{
    pfSeqMode(smoke_seq, PFSEQ_START);
    pfSwitchVal(s, 3);
}
else
    pfSwitchVal(s, PFSWITCH_OFF);
...
```

pfLOD Nodes

A pfLOD is a level-of-detail node. Level-of-detail switching is an advanced concept that is discussed in Chapter 7, “Frame and Load Control.” A level-of-detail node specifies how its children are to be displayed, based on the visual range from the channel’s viewpoint. Each child has a defined range, and the entire pfLOD has a defined center.

Table 5-5 describes the functions for working with pfLODs.

Table 5-5 pfLOD Functions

Function	Description
pfNewLOD	Create a level of detail node.
pfLODRange	Set a range at which to use a specified child node.
pfGetLODRange	Find out the range for a given node.
pfLODCenter	Set the pfLOD center.
pfGetLODCenter	Retrieve the pfLOD center.

Table 5-5 (continued) pfLOD Functions

Function	Description
pfLODTransition	Set the width of a specified transition.
pfGetLODTransition	Get the width of a specified transition.

pfLayer Nodes

A pfLayer is a leaf node that resolves the visual priority of coplanar geometry. A pfLayer allows the application to define a set of *base geometry* and a set of *layer geometry* (sometimes called *decal geometry*). The base geometry and the decal geometry should be coplanar, and the decal geometry must lie within the extent of the base polygons.

Table 5-6 describes the functions for working with pfLayers.

Table 5-6 pfLayer Functions

Function	Description
pfNewLayer	Create a pfLayer node.
pfLayerMode	Specify a hardware mode to use in drawing decals.
pfGetLayerMode	Get current mode.
pfLayerBase	Specify the child containing base geometry.
pfGetLayerBase	Find out which child contains base geometry.
pfLayerDecal	Specify the child containing decal geometry.
pfGetLayerDecal	Find out which child contains decal geometry.

pfLayer nodes can be used to overlay any sort of markings on a given polygon and are important to avoid *flimmering*. Example 5-6 demonstrates how to display runway markings as a decal above of a coplanar runway. This example uses the performance mode PFDECAL_BASE_FAST for layering; as described in the reference page, other available modes are PFDECAL_BASE_HIGH_QUALITY, PFDECAL_BASE_DISPLACE, and PFDECAL_BASE_STENCIL.

Example 5-6 Marking a Runway With a pfLayer Node

```
pfLayer *layer;
pfGeode *runway, *runway_markings;

...
/* avoid flimmering of runway and runway_markings */
layer = pfNewLayer();
pfLayerBase(layer, runway);
pfLayerDecal(layer, runway_markings);
pfLayerMode(layer, PFDECAL_BASE_FAST);
```

pfLightPoint Nodes

A pfLightPoint is a leaf node that represents a light point or set of light points. Light points don't provide overall illumination for a scene; rather, they are discrete points that emanate light, such as street lights and runway edge lights. A pfLightPoint can contain one or several light points that share common attributes, such as color, intensity, direction, and shape.

The pfLPointState mechanism provides a more comprehensive approach to light point simulation. pfLightPoint nodes are retained only for backward compatibility with previous releases of IRIS Performer. New applications should use pfLPointStates.

Table 5-7 describes the functions for working with pfLightPoints.

Table 5-7 pfLightPoint Functions

Function	Description
pfNewLPoint	Create a new pfLightPoint node.
pfLPointSize	Set the size of the points in a node.
pfGetLPointSize	Find out the size of the points in a node.
pfLPointColor	Set the color of a specified point.
pfGetLPointColor	Find out the color of a point.
pfLPointRot	Set the direction and rotation.
pfGetLPointRot	Find out the direction a point is facing.

Table 5-7 (continued) pfLightPoint Functions

Function	Description
pfLPointShape	Set the horizontal and vertical envelope.
pfGetLPointShape	Find out the shape of a point.
pfLPointPos	Set the position of each point.
pfGetLPointPos	Find out the position of a point.

Example 5-7 demonstrates some operations on light points.

Example 5-7 Setting Up Light Points

```
pfLightPoint *lp;
pfVec4 red;
pfVec3 pos;
...
/* create a string of ten light points */
lp = pfNewLPoint(10);

/* set the size in screen pixels for each light */
pfLPointSize(lp, 1.0f);

/* set the light point color */
pfSetVec4(red, 1.0f, 0.0f, 0.0f, 1.0f);
pfLPointColor(lp, red);

/* set the direction the light points face */
pfLPointRot(lp, 270.0f, 30.0f, 0.0f);

/* Set the position of light point 5 */
pfSetVec3(pos,10.0f,10.0f,0.0f);
pfLPointPos(lp, 5, pos);
...
```

pfLightSource Nodes

A pfLightSource, unlike a pfLightPoint, provides light for a scene's geometry but is itself invisible.

Illuminating the geometry in a scene graph requires a light source which the graphics hardware uses to compute surface shading. While the *libpr* primitive pfLight (see “Lighting” on page 348) provides a hardware light source, it isn’t a pfNode and so cannot benefit from the features provided by a scene graph, such as transformation hierarchy, switches, and sequenced animations.

The pfLightSource node allows you to add lighting information to your scene graph. pfLightSource is multiply inherited; it’s derived from both pfNode and pfLight, so you can use a pfLightSource* as an argument to a function that requires a pfNode* or pfLight*, as illustrated in Example 5-8.

Example 5-8 pfLightSource Pointers and Multiple Inheritance

```
pfLightSource *lsource = pfNewLSource();

/* Set color to red */
pfLightColor(lsource, 1.0f, 0.0f, 0.0f);

/* Add light source to scene */
pfAddChild(scene, lsource);
```

In this example *lsource* is the pfLight* argument to **pfLightColor()** and the pfNode* argument to **pfAddChild()**.

The scope, or area of influence, of a pfLightSource is global and isn’t affected by its location in the scene graph unless it is culled during the cull traversal. If not culled, it illuminates everything in the pfScene of which it is a member and doesn’t affect anything outside the pfScene besides custom geometry rendered by the application in a channel draw callback.

A pfLightSource does inherit transformations and switch values from its parents in the scene graph. Thus you can attach a light source to a moving object and easily turn it on and off. Example 5-9 shows how you might configure the headlights of a moving car:

Example 5-9 Car Headlights as pfLightSource Nodes

```
static pfNode* initCarGeometry(void);

    pfDCS *car = pfNewDCS();
    pfLightSource *leftHeadlight = pfNewLSource();
    pfLightSource *rightHeadlight = pfNewLSource();
```

```
/*
 * Set white light color. This isn't really necessary
 * since pfLightSources are white by default.
 */
pfLightColor(leftHeadlight, 1.0f, 1.0f, 1.0f);
pfLightColor(rightHeadlight, 1.0f, 1.0f, 1.0f);

/* Position headlights on front of car as local light
 * sources. */
pfLightPos(leftHeadlight, -1.0f, 0.0f, 0.0f, 1.0f);
pfLightPos(rightHeadlight, 1.0f, 0.0f, 0.0f, 1.0f);

/*
 * Configure headlights as spotlights pointing in same
 * direction as car: +Y direction. Set the spotlights
 * to have falloff of 1 and angular spread of 30 degrees.
 */
pfSpotLightDir(leftHeadlight, 0.0f, 1.0f, 0.0f);
pfSpotLightCone(leftHeadlight, 1.0f, 30.0f);
pfSpotLightDir(rightHeadlight, 0.0f, 1.0f, 0.0f);
pfSpotLightCone(rightHeadlight, 1.0f, 30.0f);

/*
 * Turn on headlights. This isn't really necessary since
 * pfLightSources are on by default.
 */
pfLightOn(leftHeadlight);
pfLightOn(rightHeadlight);

/* Create car geometry and add to car DCS */
pfAddChild(car, initCarGeometry());

/* Attach headlights to moving car. */
pfAddChild(car, leftHeadlight);
pfAddChild(car, rightHeadlight);
```

In Example 5-9, both the position and spotlight direction of the light sources are transformed by the combination of the car's pfDCS and all other pfDCS or pfSCS nodes above it in the scene graph. Note that we specified to **pfLightPos()** that the pfLightSource have a location with the fourth coordinate nonzero. This defines a local light source; an infinite light source has zero for its fourth coordinate. An infinite light source has only a direction, not a location; it emits parallel rays because it is considered

infinitely far away. (The sun, for example, is approximated as an infinite light source.) Transformations change only the direction of an infinite light source, not its position.

Also note in Example 5-9 that a pfSwitch isn't necessary to turn a pfLightSource on or off; you can directly switch a pfLightSource with **pfLightOn()** and **pfLightOff()**.

pfLightSources are treated specially when drawing a scene graph. All paths leading to pfLightSources in a scene are traversed before the scene is traversed. (For information about paths, see "Paths Through the Scene Graph" in Chapter 6.) This special, initial traversal guarantees that the graphics hardware will be configured with all pfLightSources so that they will affect all geometry in the scene. In addition, these light sources are set up before the channel draw callback is invoked, so they affect all custom geometry rendered by the application both before and after **pfDraw()**. (See "pfNode Cull and Draw Callbacks" in Chapter 6 for more on callbacks.)

By default, pfLightSource traversal is enabled and carried out by the cull traversal. If you wish to bypass this traversal and thereby ignore all pfLightSources in your scene, then set the PFCULL_IGNORE_LSOURCES bit in the mode passed to **pfChanTravMode()** for the PFTRAV_CULL traversal.

A pfLightSource has no default bounding volume, though you can assign it one with **pfNodeBSphere()**. If a pfLightSource is assigned a bounding volume it becomes subject to view frustum culling and will be discarded if it is completely outside the view frustum of the pfChannel. If the light source is found to be within the view frustum or it has an empty bounding volume (the default), the pfLightSource will affect everything in view. By setting the appropriate bounding sphere it is possible to roughly specify a light source with limited influence.

Table 5-8 describes the functions for working with pfLightSources.

Table 5-8 pfLightSource Functions

Function	Description
pfNewLSource	Create a new pfLightSource node.
pfLightAmbient	Set the ambient color for a pfLight.
pfGetLightAmbient	Get the ambient color for a pfLight.
pfLightColor	Set the color for a pfLight.
pfGetLightColor	Find out the color of a pfLight.
pfSpotLightDir	Aim a spotlight in the given direction.
pfGetSpotLightDir	Determine the direction a spotlight is pointing.
pfSpotLightCone	Set the width of a spotlight cone.
pfGetSpotLightCone	Find out the width of a spotlight cone.
pfLightPos	Set the position of a light.
pfGetLightPos	Get the position of a light.
pfIsLightOn	Determine whether a light is on or off.
pfLightOn	Enable a light; execute saved-up modifications to the light.
pfLightOff	Disable a light.

pfGeode Nodes

pfGeode is short for *geometry node* and is the primary node for defining geometry in *libpf*. A pfGeode contains a list of geometry structures called pfGeoSets, which are part of the IRIS Performer *libpr* library. pfGeoSets encapsulate graphics state and geometry and are described in the "Geometry Sets" section of Chapter 10, "libpr Basics." It is important to understand that pfGeoSets are not nodes but are simply elements of a pfGeode.

Table 5-9 describes the functions for working with pfGeodes.

Table 5-9 pfGeode Functions

Function	Description
pfNewGeode	Create a pfGeode.
pfAddGSet	Add a pfGeoSet.
pfRemoveGSet	Remove a pfGeoSet.
pfInsertGSet	Insert a pfGeoSet.
pfReplaceGSet	Replace a pfGeoSet.
pfGetGSet	Supply a pointer to the specified pfGeoSet.
pfGetNumGSets	Determine how many pfGeoSets are in the given pfGeode.

Example 5-10 shows how to attach several pfGeoSets to a pfGeode.

Example 5-10 Adding pfGeoSets to a pfGeode

```
pfGeode *car1;
pfGeoSet *muffler, *frame, *windows, *seats, *tires;

muffler = read_in_muffler_geometry();
frame = read_in_frame_geometry();
seats = read_in_seat_geometry();
tires = read_in_tire_geometry();

pfAddGSet(car1, muffler);
pfAddGSet(car1, frame);
pfAddGSet(car1, seats);
...
```

pfText Nodes

A pfText node is *libpf* leaf node that contains a set of *libpr* pfStrings that should be rendered based on the libpf cull and draw traversals. In this sense a pfText is similar to a pfGeode except that it renders 3-dimensional text through the *libpr* pfString and pfFont mechanisms rather than rendering standard 3-dimensional geometry via *libpr* pfGeoSet and pfGeoState

functionality. pfText nodes are useful for displaying 3-dimensional text and other collections of geometry from a fixed index list. Table 5-10 lists the major pfText functions.

Table 5-10 pfText Functions

Function	Description
pfNewText	Create a pfText.
pfAddString	Add a pfString.
pfRemoveString	Remove a pfString.
pfInsertString	Insert a pfString.
pfReplaceString	Replace a pfString.
pfGetString	Supply a pointer to the specified pfString.
pfGetNumStrings	Determine how many pfStrings are in the given pfText.

Using the pfText facility is easy. Example 5-11 shows how a pfFont is defined, how pfStrings are created that reference that font, and then how those pfStrings are added to a pfText node for display. See the description of pfStrings and pfFonts in Chapter 10, “libpr Basics,” for information on setting up individual strings to input into a pfText node

Example 5-11 Adding pfStrings to a pfText

```
int nStrings,i;
char tmpBuf[8192];
char fontName[128];
pfFont *fnt = NULL;
/* Create a new text node
pfText *txt = pfNewText();

/* Read in font using libpfdu utility function */
scanf(“%s”,fontName);
fnt = pfdLoadFont(“type1”,fontName,PFDFONT_EXTRUDED);

/* Cant render pfText or libpr pfString without a pfFont */
if (fnt == NULL)
    pfNotify(PFNFY_WARN,PFNFY_PRINT,
             ”No Such Font - %s\n”,fontName);
```

```
/* Read nStrings text strings from standard input and */
/* Attach them to a pfText */
scanf("%d",&nStrings);
for(i=0;i<nStrings;i++)
{
    char c;
    int j=0;
    int done = 0;
    pfString *curStr = NULL;

    while(done < 2) /* READ STRING - END on '||' */
    {
        c = getchar();
        if (c == '|')
            done++;
        else
            done = 0;
        tmpBuf[j++] = c;
    }
    tmpBuf[PF_MAX2(j-2,0)] = '\0';

/* Create new libpr pfString structure to attach to pfText */
    curStr = pfNewString(pfGetSharedArena());

/* Set the font for the libpr pfString */
    pfStringFont(curStr, fnt);

/* Assign the char string to the pfString */
    pfStringString(curStr, tmpBuf);

/* Add this libpr pfString to the pfText node */
/* Like adding a libpr pfGeoSet to a pfGeode */
    pfAddString(txt, curStr);
}
pfAddChild(SceneGroup, txt);
```

pfBillboard Nodes

A pfBillboard is a pfGeode that rotates its children's geometry to follow the view direction or the eyepoint. Billboards are useful for portraying complex objects that are roughly symmetrical in one or more axes. The billboard rotates to always present the same image to the viewer using far fewer polygons than a solid model uses. In this way, billboards reduce both

transformation and pixel fill demands on the graphics subsystem at the expense of some additional host processing. A classic example is a textured billboard of a single quadrilateral representing a tree.

Because a pfBillboard is also a pfGeode, you can pass a pfBillboard argument to any pfGeode routine. To add geometry, call **pfAddGSet()** (see "pfGeode Nodes" on page 137). Each pfGeoSet in the pfBillboard is treated as a separate piece of billboard geometry; each one turns so that it always faces the eye point.

pfBillboards can be either constrained to rotate about an axis, as is done for a tree or a lamp post, or constrained only by a point, as when simulating a cloud or a puff of smoke. Specify the rotation mode by calling **pfBboardMode()**; specify the rotational axis by calling **pfBboardAxis()**. Since rotating the geometry to the eyepoint doesn't fully constrain the orientation of a point-rotating billboard, modes are available to use the additional degree of freedom to align the billboard in eye space or world space. Usually the normals of billboards are specified to be parallel to the rotational axis to avoid lighting anomalies.

pfFlatten() is highly recommended for billboards. If a billboard lies beneath a pfSCS or pfDCS, an additional transformation is done for each billboard. This can have a substantial performance impact on the cull process, where billboards are transformed.

Table 5-11 describes the functions for working with pfBillboards.

Table 5-11 pfBillboard Functions

Function	Description
pfNewBboard	Create a pfBillboard node.
pfBboardPos	Set a billboard's position.
pfGetBboardPos	Find out a billboard's position.
pfBboardAxis	Specify the rotation or alignment axis.
pfGetBboardAxis	Find out the rotation or alignment axis.
pfBboardMode	Specify a billboard's rotation type.
pfGetBboardMode	Find out a billboard's rotation type.

Example 5-12 demonstrates the construction of a pfBillboard node. The code can be found in */usr/share/Performer/src/pguide/libpf/C/billboard.c*.

Example 5-12 Setting Up a pfBillboard

```
static pfVec2 BBTexCoords[] ={{0.0f, 0.0f},
                              {1.0f, 0.0f},
                              {1.0f, 1.0f},
                              {0.0f, 1.0f}};

static pfVec3 BBVertCoords[4] = /* XZ plane for pt bboards */
                         {{-0.5f, 0.0f, 0.0f},
                          { 0.5f, 0.0f, 0.0f},
                          { 0.5f, 0.0f, 1.0f},
                          {-0.5f, 0.0f, 1.0f}};

static pfVec3 BBAxes[4] = {{1.0f, 0.0f, 0.0f}, /* X */
                           {0.0f, 1.0f, 0.0f}, /* Y */
                           {0.0f, 0.0f, 1.0f}, /* Z */
                           {0.0f, 0.0f, 1.0f}}; /*world Zup*/

static int BBPrimLens[] = { 4 };

static pfVec4 BBColors[] = {{1.0, 1.0, 1.0, 1.0}};


/* Convert static data to pfMalloc'ed data */
static void*
memdup(void *mem, size_t bytes, void *arena)
{
    void *data = pfMalloc(bytes, arena);
    memcpy(data, mem, bytes);
    return data;
}

/* For pedagogical use only. Reasonable performance
 * requires more then one pfGeoSet per pfBillboard.
 */

pfBillboard*
MakeABill(pfVec3 pos, pfGeoState *gst, long bbType)
{
    pfGeoSet *gset;
    pfGeoState *gstate;
    pfBillboard *bill;
```

```
void *arena = pfGetSharedArena();

gset = pfNewGSet(arena);
gstate = pfNewGState(arena);

pfGStateMode(gstate, PFSTATE_ENLIGHTING, PF_OFF);
pfGStateMode(gstate, PFSTATE_ENTEXTURE, PF_ON);
/*.... Create/load texture map for billboard... */
pfGStateAttr(gstate, PFSTATE_TEXTURE, texture);
pfGSetGState(gset, gstate);

pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX,
    memdup(BBVertCoords, sizeof(BBVertCoords), arena),
    NULL);
pfGSetAttr(gset, PFGS_TEXCOORD2, PFGS_PER_VERTEX,
    memdup(BBTexCoords, sizeof(BBTexCoords), arena),
    NULL);

pfGSetAttr(gset, PFGS_COLOR4, PFGS_OVERALL,
    memdup(BBColors, sizeof(BBColors), arena),
    NULL);

pfGSetPrimLengths(gset,
    (int*)memdup(BBPrimLens, sizeof(BBPrimLens), arena));
pfGSetPrimType(gset, PFGS_QUADS);
pfGSetNumPrims(gset, 1);
pfGSetGState(gset, gst);

bill = pfNewBboard();
switch (bbType)
{
case PF_X: /* axial rotate */
case PF_Y:
case PF_Z:
    pfBboardAxis(bill, BBAxes[bbType]);
    pfBboardMode(bill, PFBB_ROT, PFBB_AXIAL_ROT);
    break;
case 3: /* point rotate */
    pfBboardAxis(bill, BBAxes[bbType]);
    pfBboardMode(bill, PFBB_ROT, PFBB_POINT_ROT_WORLD);
    break;
}
pfAddGSet(bill, gset);
pfBboardPos(bill, 0, pos);
```

```
    return bill;
}
```

pfPartition Nodes

A pfPartition is a pfGroup that organizes the scene graphs of its children into a static data structure that can be more efficient for intersections. Currently, partitions are only useful for data that lies more or less on an XY plane, such as terrain. A pfPartition would therefore be inappropriate for a skyscraper model.

Partition construction comes in two phases. After a piece of the scene graph has been placed under the pfPartition, **pfBuildPart()** examines the spatial arrangement of geometry beneath the pfPartition and determines an appropriate origin and spacing for the grid. Because the search is exhaustive, this examination can be time-consuming the first time through. Once a good partitioning is determined, the search space can be restricted for future database loads using the partition attributes.

The second phase is invoked by **pfUpdatePart()**, which distributes the pfGeoSets under the pfPartition into cells in the spatial partition created by **pfBuildPart()**. **pfUpdatePart()** needs to be called if any geometry under the pfPartition node changes.

During intersection traversal, the segments in a pfSegSet (see “Intersection Requests: pfSegSets” in Chapter 6) are scan-converted onto the grid, yielding faster access to those pfGeoSets that potentially intersect the segment. A pfPartition can be made to function as a normal pfGroup during intersection traversal by OR-ing PFTRAV_IS_NO_PART into the intersection traversal mode in the pfSegSet.

Table 5-12 describes the functions for working with pfPartitions.

Table 5-12 pfPartition Functions

Function	Description
pfNewPart	Create a pfPartition.
pfPartVal	Set the desired pfPartition value.
pfGetPartVal	Find out the attributes of specified value.
pfPartAttr	Set the desired pfPartition attribute.
pfGetPartAttr	Find out the attributes of specified attribute.
pfBuildPart	Construct a spatial partitioning based on the attributes.
pfUpdatePart	Traverse the partition's children and incorporate changes.
pfGetPartType	Determine what kind of partition is being used.

Example 5-13 demonstrates setting up and using a pfPartition node.

Example 5-13 Setting Up a Partition

```
pfGroup *terrain;
pfPartition *partition;
pfScene *scene;

...

terrain = read_in_grid_aligned_terrain();

...

/* create a default partitioning of a terrain grid */
partition = pfNewPart();
pfAddChild(scene, partition);
pfAddChild(partition, terrain);
pfBuildPart(partition);

...

/* use the partitions to perform efficient intersections
 * of sets of segments with the terrain */
for(i = 0; i < numVehicles; i++)
```

```
    pfNodeIsectSegs(partition, vehicle_segment_set[i],
                    hit_struct);
...
```

Sample Program

The sample program shown in Example 5-14 demonstrates scene graph construction, shared instancing, and transformation inheritance. The program uses IRIS Performer objects and functions that are described fully in later chapters.

This program reads the names of two objects from the command line, although defaults are supplied if file names are not given. These files are loaded and a second instance of each object is created. In each case, this instance is made to orbit the original object, and the second pair are also placed in orbit around the first. This program is "*inherit.c*" and is part of the suite of IRIS Performer Programmer's Guide example programs.

Example 5-14 Inheritance demonstration program

```
/*
 * inherit.c - transform inheritance example
 */

#include <math.h>
#include <Performer/pf.h>
#include <Performer/pfdu.h>

int
main(int argc, char *argv[])
{
    pfPipe *pipe;
    pfPipeWindow *pw;
    pfScene *scene;
    pfChannel *chan;
    pfCoord view;
    float z, s, c;
    pfNode *model1, *model2;
    pfDCS *node1, *node2;
    pfDCS *dcs1, *dcs2, *dcs3, *dcs4;
    pfSphere sphere;
    char *file1, *file2;
```

```
/* choose default objects of none specified */
file1 = (argc > 1) ? argv[1] : “blob.nff”;
file2 = (argc > 1) ? argv[1] : “torus.nff”;

/* Initialize Performer */
pfInit();

pfFilePath(
“.”
“:./data”
“:../data”
“:../../data”
“:../../../data”
“:../../../../data”
“:/usr/share/Performer/data”);

/* Single thread for simplicity */
pfMultiprocess(PFMP_DEFAULT);

/* Load all loader DSO’s before pfConfig() forks */
pfdInitConverter(file1);
pfdInitConverter(file2);

/* Configure */
pfConfig();

/* Load the files */
if ((model1 = pfdLoadFile(file1)) == NULL)
{
    pfExit();
    exit(-1);
}
if ((model2 = pfdLoadFile(file2)) == NULL)
{
    pfExit();
    exit(-1);
}

/* scale models to unit size */
node1 = pfNewDCS();
pfAddChild(node1, model1);
pfGetNodeBSphere(model1, &sphere);
if (sphere.radius > 0.0f)
    pfDCSScale(node1, 1.0f/sphere.radius);
```

```
node2 = pfNewDCS();
pfAddChild(node2, model2);
pfGetNodeBSphere(model2, &sphere);
if (sphere.radius > 0.0f)
    pfDCSScale(node2, 1.0f/sphere.radius);

/* Create the hierarchy */
dcs4 = pfNewDCS();
pfAddChild(dcs4, node1);
pfDCSScale(dcs4, 0.5f);

dcs3 = pfNewDCS();
pfAddChild(dcs3, node1);
pfAddChild(dcs3, dcs4);

dcs1 = pfNewDCS();
pfAddChild(dcs1, node2);

dcs2 = pfNewDCS();
pfAddChild(dcs2, node2);
pfDCSScale(dcs2, 0.5f);
pfAddChild(dcs1, dcs2);

scene = pfNewScene();
pfAddChild(scene, dcs1);
pfAddChild(scene, dcs3);
pfAddChild(scene, pfNewLSource());

/* Configure and open GL window */
pipe = pfGetPipe(0);
pw = pfNewPWin(pipe);
pfPWinType(pw, PFPWIN_TYPE_X);
pfPWinName(pw, “IRIS Performer”);
pfPWinOriginSize(pw, 0, 0, 500, 500);
pfOpenPWin(pw);

chan = pfNewChan(pipe);
pfChanScene(chan, scene);

pfSetVec3(view.xyz, 0.0f, 0.0f, 15.0f);
pfSetVec3(view.hpr, 0.0f, -90.0f, 0.0f);
pfChanView(chan, view.xyz, view.hpr);

/* Loop through various transformations of the DCS’s */
```

```
    for (z = 0.0f; z < 1084; z += 4.0f)
    {
        pfDCSRot(dcs1,
        (z < 360) ? (int) z % 360 : 0.0f,
        (z > 360 && z < 720) ? (int) z % 360 : 0.0f,
        (z > 720) ? (int) z % 360 : 0.0f);

        pfSinCos(z, &s, &c);
        pfDCSTrans(dcs2, 1.0f * c, 1.0f * s, 0.0f);

        pfDCSRot(dcs3, z, 0, 0);
        pfDCSTrans(dcs3, 4.0f * c, 4.0f * s, 4.0f * s);
        pfDCSRot(dcs4, 0, 0, z);
        pfDCSTrans(dcs4, 1.0f * c, 1.0f * s, 0.0f);

        pfFrame();
    }

    /* show objects static for three seconds */
    sleep(3);

    pfExit();
    exit(0);
}
```

Chapter 6

“Database Traversal”

This chapter explains how to manipulate, traverse, and examine a scene graph.

Chapter 6

Database Traversal

Chapter 5, “Nodes and Node Types,” described the node types used by *libpf*. This chapter describes the operations that can be performed on the run-time database defined by a scene graph. These operations typically work with part or all of a scene graph and are known as *traversals* because they traverse the database hierarchy. IRIS Performer supports four major kinds of database traversals:

- Application
- Cull
- Draw
- Intersection

The application traversal updates the active elements in the scene graph for the next frame. This includes processing active nodes such as pfMorph and invoking user supplied callbacks for animations or other embedded behaviors.

Visual processing consists of two basic traversals: culling and drawing. The cull traversal selects the visible portions of the database and puts them into a display list. The draw traversal then runs through that display list and sends rendering commands to the Geometry Pipeline. Once you have set up all the necessary elements, culling and drawing are automatic, although you can customize each traversal for special purposes.

The intersection traversal computes the intersection of one or more line segments with the database. The intersection traversal is user-directed. Intersections are used to determine

- height above terrain
- line-of-sight visibility
- collisions with database objects

Like other traversals, intersection traversals can be directed by the application through identification masks and function callbacks. Table 6-1 lists the routines and data types relevant to each of the major traversals; more information about the listed traversal attributes can be found later in this chapter and in the appropriate reference pages.

Table 6-1 Traversal Attributes for the Major Traversals

Traversal Attribute	Application PFTRAV_APP	Cull PFTRAV_CULL	Draw PFTRAV_DRAW	Intersection PFTRAV_ISECT
Controllers	pfChannel	pfChannel	pfChannel	pfSegSet
Global Activation	pfFrame() pfSync() pfAppFrame()	pfFrame()	pfFrame()	pfFrame() pfNodeIsect-Segs(), pfChan-NodeIsectSegs()
Global Callbacks	pfChanTrav-Func()	pfChanTrav-Func()	pfChanTrav-Func()	pfIsectFunc()
Activation within Callback	pfApp()	pfCull()	pfDraw()	pfFrame() pfNodeIsect-Segs(), pfChan-NodeIsectSegs()
Path Activation	NA	pfCullPath()	NA	NA
Modes	pfChanTrav-Mode()	pfChanTrav-Mode()	pfChanTrav-Mode()	pfSegSet (also discriminator callback)
Node Callbacks	pfNodeTrav-Funcs()	pfNodeTrav-Funcs()	pfNodeTrav-Funcs()	pfNodeTrav-Funcs()
Traverser Masks	pfChanTrav-Mask()	pfChanTrav-Mask()	pfChanTrav-Mask()	pfSegSet mask
Traversee Masks	pfNodeTrav-Mask()	pfNodeTrav-Mask()	pfNodeTrav-Mask()	pfNodeTrav-Mask() pfGSetIsect-Mask()

Scene Graph Hierarchy

A visual database, also known as a *scene*, contains state information and geometry. A scene is organized into a hierarchical structure known as a *graph*. The graph is composed of connected database units called nodes. Nodes that are attached below other nodes in the tree are called *children*. Children belong to their *parent* node. Nodes with the same parent are called *siblings*.

Database Traversals

The scene hierarchy supplies definitions of how items in the database relate to one another. It contains information about the logical and spatial organization of the database. The scene hierarchy is processed by visiting the nodes in depth-first order and operating on them. The process of visiting, or touching, the nodes is called *traversing* the hierarchy. The tree is traversed from top to bottom and from left to right. IRIS Performer implements several types of database traversals, including application, clone, cull, delete, draw, flatten, and intersect. These traversals are described in more detail later in this chapter.

The principal traversals (application, cull, draw and intersect) all use a similar traversal mechanism that employs traversal masks and callbacks to control the traversal. When a node is visited during the traversal, processing is performed in the following order:

1. Prune the node based on the bitwise AND of the traversal masks of the node and the pfChannel (or pfSegSet). If pruned, traversal continues with the nodes siblings.
2. invoke the node's pre-traversal callback, if any, and either prune, continue, or terminate the traversal, depending on callback's return value.
3. Traverse, beginning again at step 1, the node's children or geometry (pfGeoSets). If the node is a pfSwitch, a pfSequence, or a pfLOD, the state of the node affects which children are traversed.
4. Invoke the node's post-traversal callback, if any.

State Inheritance

In addition to imposing a logical and spatial ordering of the database, the hierarchy also defines how state is inherited between parent and child nodes during scene graph traversals. For example, a parent node that represents a transformation causes the subsequent transformation of each of its children when it and they are traversed. In other words, the children inherit *state*, which includes the current coordinate transformation, from their parent node during database traversal.

A *transformation* is a 4x4 homogeneous matrix that defines a 3D transformation of geometry, which typically consist of scaling, rotation, and translation. The node types pfSCS and pfDCS both represent transformations. Transformations are inherited through the scene graph with each new transformation being concatenated onto the ones above it in the scene graph This allows chained articulations and complex modeling hierarchies.

The effects of state are propagated downward only, not from left to right nor upward. This means that only parents can affect their children—siblings have no effect on each other nor on their parents. This behavior results in an easy-to-understand hierarchy that is well suited for high-performance traversals.

Graphics state such as textures and materials are not inherited by way of the scene graph, but are encapsulated in leaf geometry nodes called pfGeode nodes, which are described in the section “Node Types” in Chapter 5.

Database Organization

IRIS Performer uses the spatial organization of the database to increase the performance of certain operations such as drawing and intersections. It is therefore recommended that you consider the spatial arrangement of your database. What you might think of as a logical arrangement of items in the database may not match the spatial arrangement of those items in the visual environment, which can reduce IRIS Performer’s ability to optimize operations on the database. See “Organizing a Database for Efficient Culling” on page 163 for more information about spatial organization in a visual database and the efficiency of database operations.

Application Traversal

The application traversal is the first traversal that occurs during the processing of the scene graph in preparation for rendering a frame. It is initiated by calling **pfAppFrame()**. If **pfAppFrame()** is not explicitly called, the traversal is automatically invoked by **pfSync()** or **pfFrame()**. An application traversal can be invoked for each channel, but usually channels share the same application traversal (see **pfChanShare()**).

The application traversal updates dynamic elements in the scene graph, such as geometric morphing carried invoked by a pfMorph node. The application traversal is also often used for implementing animations or other custom processing when it is desirable to have those behaviors embedded in the scene graph and invoked by IRIS Performer rather than requiring application code to invoke them every frame.

The traversal proceeds as described in “Database Traversals”. The selection of which children to traverse is also affected by the application traversal mode of the channel, in particular the choice of all, none or one of the children of pfLOD, pfSequence and pfSwitch nodes is possible. By default, the traversal obeys the current selection dictated by these nodes.

The following example from the Open Inventor loader (this loader reads both Open Inventor and VRML files) shows a simple callback changing the transformation on a pfDCS every frame. This and other examples can be found in */usr/share/Performer/src/lib/libpfiv/pfiv.C*.

Example 6-1 Application Callback to Make a Pendulum

```
int
AttachPendulum(pfDCS *dcs, PendulumData *pd)
{
  pfNodeTravFuncs(dcs, PFTRAV_APP, PendulumFunc, NULL);
  pfNodeTravData(dcs, PFTRAV_APP, pd);
}

int
PendulumFunc(pfTraverser *trav, void *userData)
{
  PendulumData *pd = (PendulumData*)userData;
  pfDCS *dcs = (pfDCS*)pfGetTravNode(trav);

  if (pd->on)
```

```
  {
    pfMatrix mat;
    double now = pfGetFrameTimeStamp();
    float frac, dummy;

    pd->lastAngle += (now -
      pd->lastTime)*360.0f*pd->frequency;
    if (pd->lastAngle > 360.0f)
      pd->lastAngle -= 360.0f;

    // using sinusoidally generated angle
    pfSinCos(pd->lastAngle, &frac, &dummy);
    frac = 0.5f + 0.5f * frac;
    frac = (1.0f - frac)*pd->angle0 + frac*pd->angle1;

    pfMakeRotMat(mat,
      frac, pd->axis[0], pd->axis[1], pd->axis[2]);
    pfDCSMat(dcs, mat);
    pd->lastTime = now;
  }

  return PFTRAV_CONT;
}
```

Cull Traversal

The cull traversal occurs in the cull phase of the *libpf* rendering pipeline and is initiated by calling **pfFrame()**. A cull traversal is performed for each pfChannel and determines the portion of the scene to be rendered. The traversal processes the subgraphs of the scene that are both visible and selected by nodes in the scene graph that control traversal (e.g. pfLOD, pfSequence, pfSwitch). The visibility culling itself is performed by testing bounding volumes in the scene graph against the channel's viewing frustum.

For customizing the cull traversal, *libpf* provides traversal masks and function callbacks for each node in the database, as well as a function callback in which the application can do its own culling of custom data structures.

Traversal Order

The cull is a depth-first traversal of the database hierarchy beginning at a pfScene, which is the hierarchy's root node. For each node, a series of tests is made to determine whether the traversal should *prune* the node—that is, eliminate it from further consideration—or continue on to that node's children. The cull traversal processing is much as described earlier, in particular the draw traversal masks are compared and the node is checked for visibility before the traversal continues on to the nodes children:

1. Prune the node, based on the channel's draw traversal mask and the node's draw mask.
2. Invoke the node's pre-cull callback and either prune, continue, or terminate the traversal, depending on callback's return value.
3. Prune the node if its bounding volume is completely outside the viewing frustum.
4. Traverse, beginning again at step 1, the node's children or geometry (pfGeoSets) if the node is completely or partially in the viewing frustum. If the node is a pfSwitch, a pfSequence, or a pfLOD, the state of the node affects which children are traversed.
5. Invoke the node's post-cull callback.

The following sections discuss these steps in more detail.

Visibility Culling

Culling determines whether a node is within a pfChannel's viewing frustum for the current frame. Nodes that are not visible are pruned—omitted from the list of objects to be drawn—so that the Geometry Pipeline doesn't waste time processing primitives that couldn't possibly appear in the final image.

Hierarchical Bounding Volumes

Testing a node for visibility compares the *bounding volume* of each object in the scene against a viewing frustum that is bounded by the near and far clip planes and the four sides of the viewing pyramid. Both nodes (see Chapter 5) and pfGeoSets (see Chapter 10) have bounding volumes that surround the geometry that they contain. Bounding volumes are simple

geometric shapes whose centers and edges are easy to locate. Bounding volumes are organized hierarchically so that the bounding volume of a parent encloses the bounding volumes of all its children. You can specify bounding volumes or let IRIS Performer generate them for you (see “Bounding Volumes” in Chapter 5).

Figure 6-1 shows a frustum and three objects surrounded by bounding boxes. Two of the objects are outside the frustum; one is within it. One of the objects outside the frustum has a bounding box whose edges intersect the frustum, as shown by the shaded area. The visibility test for this object returns TRUE, because its bounding box does intersect the view frustum even though the object itself doesn’t.

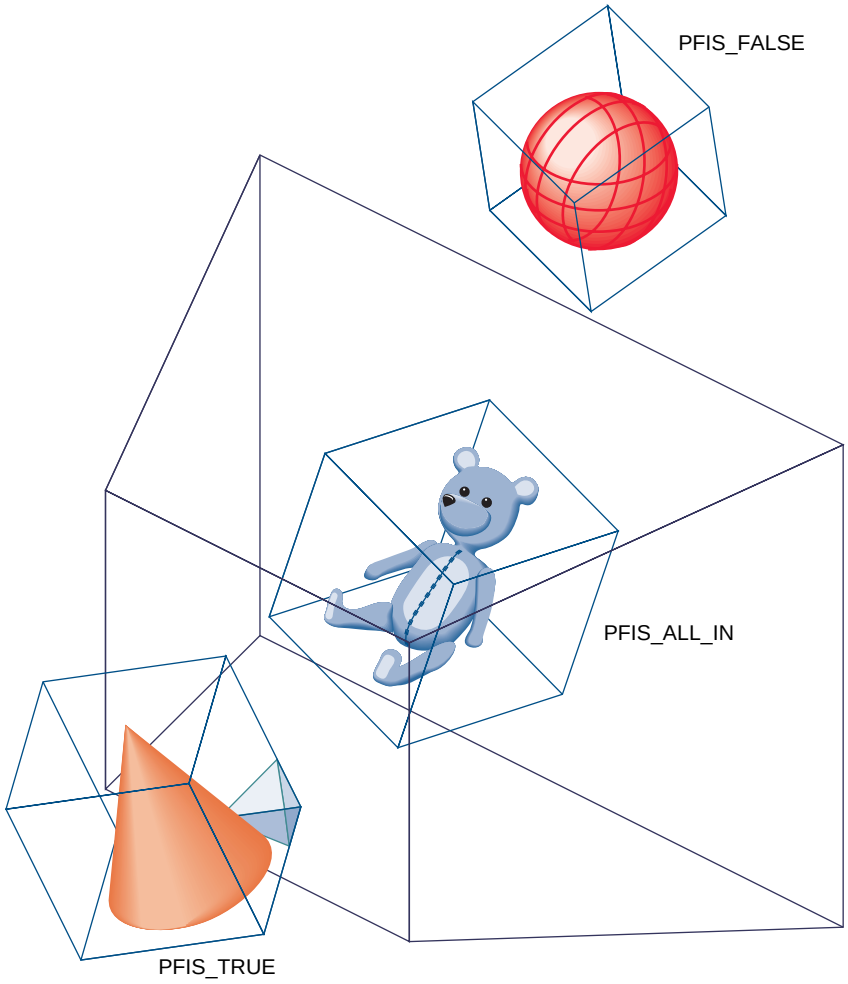


Figure 6-1 Culling to the Frustum

Visibility Testing

The cull traversal begins at the root node of a channel's scene graph (the pfScene node) and continues downward, directed by the results of the cull test at each node. At each node the cull test determines the relationship of the node's bounding volume to the viewing frustum. Possible results are that the bounding volume is entirely outside, is entirely within, is partially within, or completely contains the viewing frustum.

If the intersection test indicates that the bounding volume is entirely outside the frustum, the traversal prunes that node—that is, it doesn't consider the children of that node and continues with the node's next sibling.

If the intersection test indicates that the bounding volume is entirely inside the frustum, the node's children are not cull tested because the hierarchical nature of bounding volumes implies that the children must also be entirely within the frustum.

If the intersection test indicates that the bounding volume is partially within the frustum, or that the bounding volume completely contains the frustum, the testing process continues with the children of that node. Because a bounding volume is larger than the object it surrounds, it is possible for a bounding volume to be partially within a frustum even when none of its enclosed geometry is visible.

By default, IRIS Performer tests bounding volumes all the way down to the pfGeoSet level (see Chapter 10, "libpr Basics") to provide fine-grained culling. However, if your application is spending too much time culling, you can stop culling at the pfGeode level by calling **pfChanTravMode()**. Then if part of a pfGeode is potentially visible, all geometry in that pfGeode is drawn without cull-testing it first.

Visibility Culling Example

Figure 6-2 portrays a simple database that contains a toy block, train, and car. The block is outside the frustum, the bounding volume of the car is partially inside the frustum, and the train is completely inside the frustum.

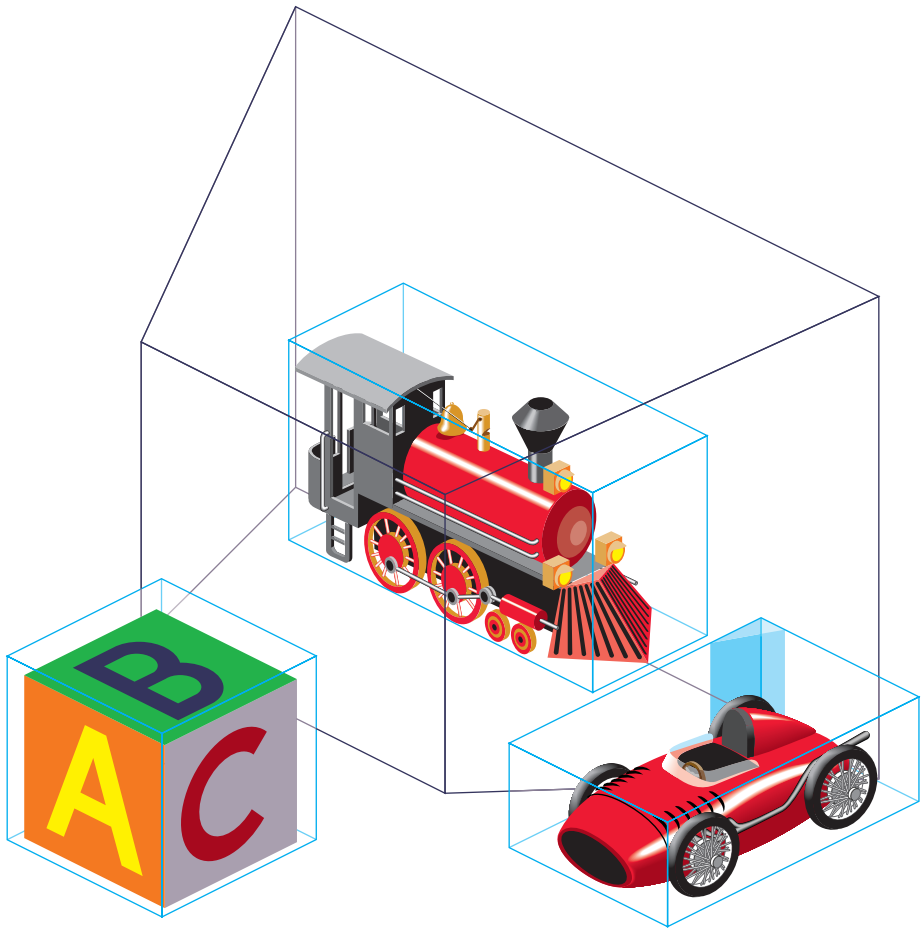


Figure 6-2 Sample Database Objects and Bounding Volumes

Organizing a Database for Efficient Culling

Efficient culling depends on having a database whose hierarchy is organized spatially. A good technique is to partition the database into regions, called *tiles*. Tiling is also required for *database paging*. Instead of culling the entire database, only the tiles that are within the view frustum need to be traversed.

The worst case for the cull traversal performance is to have a very flat hierarchy—that is, a pfScene with all the pfGeodes directly under it and many pfGeoSets in each pfGeode—or a hierarchy that is organized by object type (for example, having all trees in the database grouped under one pine tree node, rather than arranged spatially).

Figure 6-3 shows a sample database represented by cubes, cones, pyramids, and spheres. Organizing this database spatially, rather than by object type, promotes efficient culling. This type of spatial organization is the most effective control you have over efficient traversal.

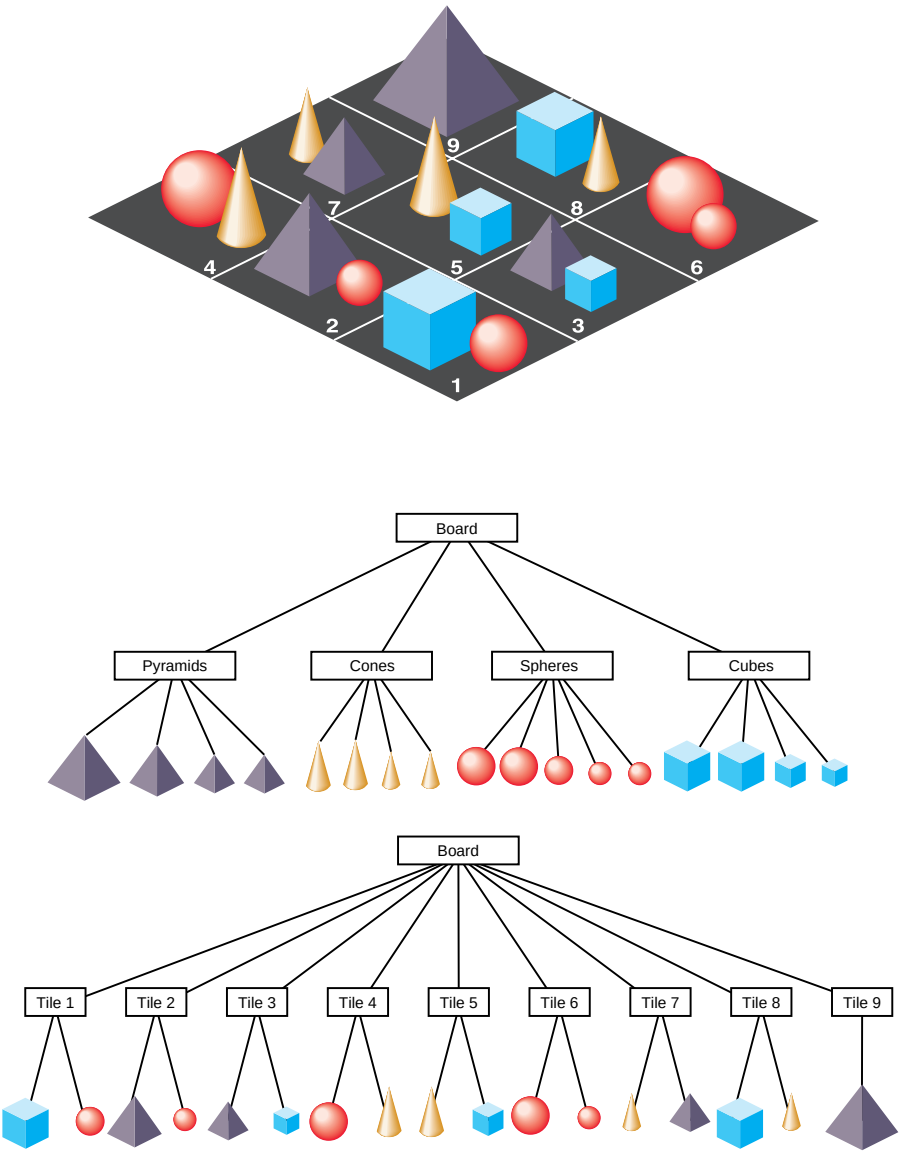


Figure 6-3 How to Partition a Database for Maximum Efficiency

When modeling a database, you should consider other trade-offs as well. Small amounts of geometry in each culling unit, whether pfGeode or pfGeoSet, provide better culling resolution and result in sending less non-visible geometry to the pipeline. Small pieces also improve the performance of line-segment intersection inquiries (see“Database Concerns” in Chapter 13). However, using many small pieces of geometry can increase the traversal time and can also reduce drawing performance. The optimal amount of geometry to place in each pfGeoSet depends on the application, database, system CPU, and graphics hardware.

Custom Visibility Culling

Existence within the frustum isn’t the only criterion that determines an object’s visibility. The item may be too distant to be seen from the viewpoint, or it may be obscured by other objects between it and the viewer, such as a wall or a hill. Atmospheric conditions can also affect object visibility. An object that is normally visible at a certain distance may not be visible at that same distance in dense fog.

Implementing more sophisticated culling requires knowledge of visibility conditions and control over the cull traversal. The cull traversal can be controlled through traversal masks, which are described in the section titled “Controlling and Customizing Traversals.”

Knowing whether an object is visible requires either prior information about the spatial organization of a database, such as cell-to-cell visibilities, or run-time testing such as computing line-of-sight visibility (LOS). You can compute simple LOS visibility by intersecting line segments that start at the eyepoint with the database. See the “Intersection Traversal” section of this chapter.

Sorting the Scene

During the cull traversal, a pfChannel can rearrange the order in which pfGeoSets are rendered for improved performance and image quality. It does this by *binning* and *sorting*. Binning is the act of placing pfGeoSets into specific *bins* which are rendered in a specific order. IRIS Performer provides two default bins: one for opaque geometry and one for blended, transparent geometry. The opaque bin is drawn before the transparent bin so transparent

surfaces are properly blended with the background scene. Applications are free to add new bins and specify arbitrary bin orderings.

Sorting is done on a per-bin basis. pfGeoSets within a given bin are sorted by a specific criterion. Two useful criteria provided by IRIS Performer are sorting by graphics state and sorting by range. When sorting by state, pfGeoSets are sorted first by their pfGeoState, then by an application-specified hierarchy of state modes, values, and attributes which are identified by PFSTATE_* tokens and are described in the *libpr* chapter. State sorting can offer a huge performance advantage since it greatly reduces the number of mode changes carried out by the Geometry Pipeline. State sorting is the default sorting configuration for the opaque bin.

Range sorting is required for proper rendering of blended, transparent surfaces which must be rendered in back-to-front order so that each surface is properly blended with the current background color. Front-to-back sorting is also supported. The default sorting for the transparent bin is back-to-front sorting (Note that the sorting granularity is per-pfGeoSet, not per-triangle so transparency sorting is not perfect).

pfChannel bins are given rendering order and sorting configuration with **pfChanBinOrder()** and **pfChanBinSort()** respectively. A bin's order is simply an integer identifying its place in the list of bins. An order less than 0 or PFSORT_NO_ORDER means that pfGeoSets which fall into the bin are drawn immediately without any ordering or sorting. Multiple bins may have the same order but the rendering precedence among these bins is undefined.

A bin's sorting configuration is given as a token identifying the major sorting criterion and then an optional list of tokens, terminated with the PFSORT_END token, that defines a state sorting hierarchy.

PFSORT_BY_STATE
: pfGeoSets are sorted first by pfGeoState then by the state elements found between the PFSORT_STATE_BGN and PFSORT_STATE_END tokens, for example.

PFSORT_FRONT_TO_BACK
: pfGeoSets are sorted by nearest to farthest range from the eyepoint. Range is computed from eyepoint to the center of the pfGeoSet's bounding volume.

PFSORT_BACK_TO_FRONT
: pfGeoSets are sorted by farthest to nearest range from the eyepoint. Range is computed from eyepoint to the center of the pfGeoSet's bounding volume.

PFSORT_QUICK
: A special, low-cost sorting technique. pfGeoSets must fall into a bin whose order is 0 in which case they will be sorted by pfGeoState and drawn immediately. This is the default sorting mode for the PFSORT_OPAQUE_BIN bin.

For example, the specification:

```
static int sort[] = {PFSORT_STATE_BGN,
                     PFSTATE_TEXTURE, PFSTATE_FRONTMTL,
                     PFSORT_STATE_END, PFSORT_END};
pfChanBinSort(chan, PFSORT_OPAQUE_BIN, PFSORT_BY_STATE,
              sort);
```

will sort the opaque bin by pfGeoState, then by pfTexture, then by pfMaterial.

A pfGeoSet's draw bin may be set directly by the application with **pfGSetDrawBin()**. Otherwise IRIS Performer automatically determines if the pfGeoSet belongs in the default opaque or transparent bins.

Paths Through the Scene Graph

You can define a chain, or *path*, of nodes in a scene graph using the pfPath data structure. (Note that a pfPath has nothing to do with filesystem paths as specified with the PFPATH environment variable or with specifying a path for a user to travel through a scene.) Once you've specified a pfPath with a call to **pfNewPath()**, you can traverse and cull that path as a subset of the entire scene graph using **pfCullPath()**. Call he function **pfCullPath()** must only from the cull callback function set by **pfChanTravFunc()**—see "Process Callbacks" on page 174 for details. For more information about the pfPath structure, see the pfPath(3pf) and pfList(3pf) reference pages.

When IRIS Performer looks for intersections, it can return a pfPath to the node containing the intersection. This feature is particularly useful when you're using instancing, in which case you can't use **pfGetParent()** to find

out where in the scene graph the given node is. Finding out the pfPath to a given node is also useful in implementing picking—for an example, see the source code in */usr/share/Performer/src/sample/apps/C/pickfly/picking.c*, or execute */usr/share/Performer/bin/pickfly* if it is installed on your system. When you click on an object in a scene, this program prints the path to the node you've picked.

Draw Traversal

The cull traversal generates a *libpr* display list of geometry and state commands (see "Display Lists" in Chapter 10), which describes the scene that is visible from a pfChannel. The draw traversal simply traverses the display list and sends commands to the Geometry Pipeline to generate the image.

Traversing a pfDispList is much faster than traversing the database hierarchy because the pfDispList *flattens* the hierarchy into a simple, efficient structure. In this way, the cull traversal removes much of the processing burden from the draw traversal; throughput greatly increases when both traversals are running in parallel.

Controlling and Customizing Traversals

The result of the cull traversal is a display list of geometry to be rendered by the draw traversal. What gets placed in the display list is determined by both visibility and by other user-specified modes and tests.

pfChannel Traversal Modes

The PFTRAV_CULL argument to **pfChanTravMode()** modifies the culling traversal. The cull mode is a bitmask that specifies the modes to enable, it is formed by the logical OR of one or more of these tokens:

- PFCULL_VIEW
- PFCULL_GSET

- PFCULL_SORT
- PFCULL_IGNORE_LSOURCES

Culling to the view frustum is enabled by PFCULL_VIEW. Culling to the pfGeoSet-level is enabled by PFCULL_GSET and can produce a tighter cull that improves rendering performance at the expense of culling time.

PFCULL_SORT causes the cull to sort geometry by state—for example, by texture or by material, in order to optimize rendering performance. It also causes transparent geometry to be drawn after opaque geometry for proper transparency effects.

By default, the enabled culling modes are PFCULL_VIEW | PFCULL_GSET | PFCULL_SORT. It is recommended that these modes be enabled unless the cull traversal becomes a significant bottleneck in the processing pipeline. In this case, try disabling PFCULL_GSET first, then PFCULL_SORT.

Normally, a pfChannel's cull traversal pre-traverses the scene, following all paths from the scene to all pfLightSources in the scene so that light sources can be set up before the normal scene traversal. If you with to disable this pre-traversal, set the PFCULL_IGNORE_LSOURCES cull enable bit but your pfLightSources will not illuminate the scene.

The PFTRAV_DRAW argument to **pfChanTravMode()** modifies the draw traversal. A mode of PFDRAW_ON is the default and will cause the pfChannel to be rendered. A mode of PFDRAW_OFF indicates that the pfChannel should not be drawn and essentially turns off the pfChannel.

pfNode Draw Mask

Each node in the database hierarchy can be assigned a mask that dictates whether the node is added to the display list and thereby whether it is drawn. This mask is called the *draw mask* (even though it is evaluated in the cull traversal) because it tells the cull process whether the node is drawable or not.

The draw mask of a node is set with **pfNodeTravMask()**. The channel also has a draw mask, which you set with **pfChanTravMask()**. By default, the masks are all 1s, or 0xffffffff.

Before testing a node for visibility, the cull traversal ANDs the two masks together. If the result is zero, the cull prunes the node. If the result is nonzero, the cull proceeds normally. Mask testing occurs before all visibility testing and function callbacks.

Masks allow you to draw different subgraphs of the scene on different channels, to turn portions of the scene graph on and off, or to ignore hidden portions of the scene graph while drawing but make them active during intersection testing.

pfNode Cull and Draw Callbacks

One of the primary mechanisms for extending IRIS Performer is through the use of function callbacks, which can be specified on a per-node basis. IRIS Performer allows separate cull and draw callbacks, which are invoked both before and after node processing. Node callbacks are set with **pfNodeTravFuncs()**.

Cull callbacks can direct the cull traversal, while draw callbacks are added to the display list and later executed in the draw traversal for custom rendering. There are pre-cull and pre-draw callbacks, invoked before a node is processed, and post-cull and post-draw callbacks, invoked after the node is processed.

The cull callbacks return a value indicating how the cull traversal should proceed, as shown in Table 6-2.

Table 6-2 Cull Callback Return Values

Value	Action
PFTRAV_CONT	Continue and traverse the children of this node.
PFTRAV_PRUNE	Skip the subgraph rooted at this node and continue.
PFTRAV_TERM	Terminate the entire traversal.

Callbacks are processed by the cull traversal in the following order:

1. If a pre-cull callback is defined, then call the pre-cull callback to get a cull result and find out whether traversal should continue. Possible return values are listed in Table 6-2.
2. If the pre-cull callback returns PFTRAV_PRUNE, the traversal returns to the parent and continues with the node's siblings, if any. If the callback returns PFTRAV_TERM, the traversal terminates immediately. Otherwise, cull processing continues.
3. If the pre-cull callback doesn't set the cull result using **pfCullResult()**, and view-frustum culling is enabled, then perform the standard node-within-frustum test and set the cull result accordingly.
4. If the cull result is PFIS_FALSE, skip the traversal of children. The post-cull callback is invoked and traversal returns so that the parent node can traverse any siblings.
5. If a pre-draw callback is defined, then place a *libpr* display-list packet in the display list so that the node's pre-draw callback will be called by the draw process. If running a combined CULLDRAW traversal, invoke the pre-draw callback directly instead.
6. Process the node, continuing the cull traversal with each of the node's children or adding the node's geometry to a display list (for pfGeodes). If the cull result was PFIS_ALL_IN, view-frustum culling is disabled during the traversal of the children.
7. If a post-draw callback is defined, then place a *libpr* display-list packet in the display list so that the node's post-draw callback will be called by the draw process. If running a combined CULLDRAW traversal, invoke the post-draw callback directly instead.
8. If a post-cull callback is defined, then call the post-cull callback.

Draw callbacks are commonly used to place tags or change state while a subgraph is rendered. Note that if the pre-draw callback is called, the post-draw callback is guaranteed to be invoked. This way the callback can restore any state modified by the pre-draw callback. This is useful for state changes such as **pfPushMatrix()** and **pfPopMatrix()**, as shown in the environment-mapping code that's part of Example 6-2.

For doing customized culling, the pre-cull callback can determine whether a PFIS_ALL_IN has already turned off view-frustum culling using **pfGetParentCullResult()**, in which case it may not wish to do its own cull testing. It can also find out the result of the standard cull test by calling **pfGetCullResult()**.

Cull callbacks can also be used to render geometry (pfGeoSets) or change graphics state. Any *libpr* drawing commands are captured in a display list and are later executed during the draw traversal (see "Display Lists" in Chapter 10). However, direct graphics library calls can be made safely only in draw function callbacks, because only the draw process of multiprocess IRIS Performer configurations is known to be associated with a window.

Example 6-2 shows some sample node callbacks.

Example 6-2 pfNode Draw Callbacks

```
void
LoadScene(char *filename)
{
    pfScene *scene = pfNewScene();
    pfGroup *root = pfNewGroup();
    pfGroup *reflectiveGeodes = NULL;

    root = pfdLoadFile(filename);
    ...
    reflectiveGeodes =
           ReturnListofGeodesWithReflectiveMaterials(root);

    /* Use a node callback in the Draw process to turn on
     * and off graphics library environment mapping before
     * and after drawing all of the pfGeodes that have
     * pfGeoStates with reflective materials.
     */
    pfNodeTravFuncs(reflectiveGeodes, PFTRAV_DRAW,
                    pfdPreDrawReflMap, pfdPostDrawReflMap);
}

/* This callback turns on graphics library environment
 * mapping.  Because it changes graphics state it must be a
 * Draw process node callback. */
long
pfdPreDrawReflMap(pfTraverser *trav, void *data)
{
```

```
    texgen(TX_S, TG_SPHEREMAP, 0);
    texgen(TX_T, TG_SPHEREMAP, 0);
    texgen(TX_S, TG_ON, NULL);
    texgen(TX_T, TG_ON, NULL);
    return NULL;
}

/* This callback turns off graphics library environment
 * mapping.  Because it also changes graphics state it also
 * must be a Draw process node callback.  Also notice that
 * it is important to return the graphics library's state to
 * the state at which it was in before the preNode callback
 * was even made.
 */
long
pfdPostDrawReflMap(pfTraverser *trav, void *data)
{
    texgen(TX_S, TG_OFF, NULL);
    texgen(TX_T, TG_OFF, NULL);
    return NULL;
}
```

Process Callbacks

libpf processes a visual database with a software-rendering pipeline composed of application, cull, and draw stages. The system of *process callbacks* allows you to insert your own custom culling and drawing functions into the rendering pipeline. Furthermore, these callbacks are invoked by the proper process when your IRIS Performer application is configured for multiprocessing.

By default, IRIS Performer culls and draws all active pfChannels when **pfFrame()** is called. However, you can specify cull and draw function callbacks so that **pfFrame()** will cause IRIS Performer to call your custom functions instead. These functions have the option of using the default IRIS Performer processing in addition to their own custom processing.

When multiprocessing is used, the rendering pipeline works on multiple frames at once. For example, when the draw process is rendering frame *n*, the cull process is working on frame *n+1*, and the application process is

working on frame *n+2*. This situation requires careful management of data so that data generated by the application is propagated to the cull process and then to the draw process at the right time. IRIS Performer manages data that is passed to the process callbacks to ensure that the data is frame-coherent and isn't corrupted.

Example 6-3 illustrates the use of a cull-process callback.

Example 6-3 Cull-Process Callbacks

```
InitChannels()
{
    ...
    /* create and configure all channels*/
    ...
    /* define callbacks for cull and draw processes */
    pfChanTravFunc(chan, PFTRAV_CULL, CullFunc);
    pfChanTravFunc(chan, PFTRAV_DRAW, DrawFunc);
    ...
}

/* The Cull callback.  Any work that needs to be done in the
 * Cull process should happen in this function.
 */
void
CullFunc(pfChannel * chan, void *data)
{
    static long first = 1;

    /* Lock down whatever processor the cull is using when
     * the cull callback is first called.
     */
    if (first)
    {
        if ((pfGetMultiprocess() & PFMP_FORK_CULL) &&
            (ViewState->procLock & PFMP_FORK_CULL))
            pfuLockDownCull(pfGetChanPipe(chan));
        first = 0;
    }

    /* User-defined pre-cull processing.  Application-
     * specific cull knowledge might be used to provide
     * things like line-of-sight culling.
     */
    PreCull(chan, data);
```

```
    /* standard Performer culling to the viewing frustum */
    pfCull();

    /* User-defined post-cull processing; this routine might
     * be used to do things like record cull state from this
     * cull to be used in future culls.
     */
    PostCull(chan, data);
}

/* The draw function callback.  I/O with IRIS GL devices must
 * happen here.  Any graphics library functionality outside
 * IRIS Performer must be done here.
 */
void
DrawFunc(pfChannel *chan, void *data)
{
    /* pre-Draw tasks like clearing the viewport */
    PreDraw(chan, data);

    pfDraw();               /* render the frame */

    /* draw HUD, read IRIS GL devices, and so on */
    PostDraw(chan, data);
}
```

Process Callbacks and Passthrough Data

Cull and draw callbacks are specified on a per-pfChannel basis using the functions **pfChanTravFunc()** with PFTRAV_CULL and PFTRAV_DRAW, respectively. **pfAllocChanData()** allocates *passthrough data*, data which is passed down the rendering pipeline to the callbacks.

In the cull phase of the rendering pipeline, IRIS Performer invokes the cull callback with a pointer to the pfChannel that is being culled and a pointer to the pfChannel's passthrough data buffer. The cull callback may modify data in the buffer. The potentially modified buffer is then copied and passed to the user's draw callback.

Default IRIS Performer processing is triggered by **pfCull()** and **pfDraw()**. By default, **pfFrame()** calls **pfCull()** first, then calls **pfDraw()**. If process callbacks are defined, however, **pfCull()** and **pfDraw()** are not invoked automatically and must be called by the callbacks to use IRIS Performer's default processing. **pfCull()** should be called only in the cull callback; it causes IRIS Performer to cull the current channel and to generate a display list suitable for rendering.

Channels culled by **pfCull()** may be drawn in the draw callback by **pfDraw()**. It is legal for the draw callback to call **pfDraw()** more than once. Multi-pass renderings performed with multiple calls to **pfDraw()** are typical when you use accumulation buffer techniques.

When the draw callback is invoked, the window will have already been properly configured for drawing the pfChannel. Specifically, the viewport, perspective, and viewing matrices are set to their correct values. User modifications of these values are not reset by **pfDraw()**. If a draw callback is specified, IRIS Performer doesn't automatically clear the viewport; it leaves that responsibility to the application. **pfClearChan()** can be called from the draw callback to clear the channel viewport. If *chan* has a **pfEarthSky()**, then the **pfEarthSky()** is drawn. Otherwise, the viewport is cleared to black and the z-buffer is cleared to its maximum value.

You should call **pfPassChanData()** to indicate that user data should be passed through the rendering pipeline, which propagate the data downstream to cull and draw callbacks. The next call to **pfFrame()** copies the channel buffer into internal buffers, so that the application is then free to modify data in the buffer without fear of corruption. The **pfPassChanData()** function should be called only when necessary, since calling it imposes some buffer-copying overhead. In addition, passthrough data should be as small as possible to reduce the time spent copying data.

The code fragment in Example 6-4 is an example of cull and draw callbacks and the passthrough data that is used to communicate with them.

Example 6-4 Using Passthrough Data to Communicate With Callback Routines

```
typedef struct
{
    long val;
}   PassData;
```

```
void cullFunc(pfChannel *chan, void *data);
void drawFunc(pfChannel *chan, void *data);

int main()
{
    PassData   *pd;

    /* allocate passthrough data */
    pd = (PassData*)pfAllocChanData(chan,sizeof(PassData));

    /* initialize channel callbacks */
    pfChanTravFunc(chan, PFTRAV_CULL, cullFunc);
    pfChanTravFunc(chan, PFTRAV_DRAW, drawFunc);

    /* main simulation loop */
    while (1)
    {
        pfSync();
        pd->val = 0;
        pfPassChanData(chan);
        pfFrame();
    }
}

void
cullFunc(pfChannel *chan, void *data)
{
    PassData   *pd = (PassData*)data;

    pd->val++;
    pfCull();
}

void
drawFunc(pfChannel *chan, void *data)
{
    PassData   *pd = (PassData*)data;
    fprintf(stderr, "%ld\n", pd->val);
    pfClearChan(chan);
    pfDraw();
}
```

This example would, regardless of the multiprocessing mode, have the values 0, 1, and 1 for *pd->val* at the points where **pfFrame()**, **pfCull()**, and

pfDraw() are called. In this way, control data can be sent down the pipeline from the application, through the cull, and on to the draw process with frame synchronization without regard to the active multiprocessing mode.

When configured as a process separate from the draw, the cull callback should not attempt to send graphics commands to an IRIS Performer window because only the draw process is attached to the window. Callbacks should not modify the IRIS Performer database, but they can use **pfGet()** routines to inquire about database information. The draw callback should not call **swapbuffers()** (or an equivalent function when using OpenGL) because IRIS Performer must control buffer swapping in order to manage the necessary frame and channel synchronization. However, if you need special control over buffer swapping, use **pfPipeSwapFunc()** to register a function as the given pipe's buffer-swapping function. Once your function is registered, it will be called instead of **swapbuffers()** and may then invoke either of these functions.

Intersection Traversal

You can make spatial inquiries in IRIS Performer by testing the intersection of line segments with geometry in the database. For example, a single line segment pointing straight down from the eyepoint can determine your height above terrain, four such segments can simulate the four tires of a car, and segments swept out by points on a moving object can determine collisions with other objects.

Testing Line Segment Intersections

The testing of each line segment or group of spatially grouped segments requires a traversal of part or all of a scene graph. You make these inquiries using **pfNodeIsectSegs()**, which intersects the specified group of segments with the subgraph rooted at the specified node. **pfChanNodeIsectSegs()** functions similarly, but includes a channel so that the traversal can make decisions based on the level-of-detail specified by pfLOD nodes.

Intersection Requests: pfSegSets

A pfSegSet is a structure that embodies an intersection request.

```
typedef struct _pfSegSet
{
    long   mode;
    void*  userData;
    pfSeg  segs[PFIS_MAX_SEGS];
    ulong  activeMask;
    ulong  isectMask;
    void*  bound;
    long   (*discFunc)(pfHit*);
} pfSegSet;
```

The *segs* field is an array of line segments making up the query. You tell **pfNodeIsectSegs()** which segments to test with by setting the corresponding bit in the *activeMask* field. If your pfSegSet contains many closely-grouped line segments, you can specify a bounding volume using the data structure's *bound* field. **pfNodeIsectSegs()** can use that bounding volume to more quickly test the request against bounding volumes in the scene graph. The *userData* field is a pointer with which you can point to other information about the request that you might want access to in a callback. The other fields are described below. The pfSegSet isn't modified during the traversal.

Intersection Return Data: pfHit Objects

Intersection information is returned in pfHit objects. These can be queried using **pfQueryHit()** and **pfMQueryHit()**. Table 6-3 lists the items that can be queried from a pfHit object.

Table 6-3 Intersection-Query Token Names

Query Token	Description
PFQHIT_FLAGS	Status and validity information
PFQHIT_SEGNUM	Index of the segment in pfSegSet request
PFQHIT_SEG	Line segment as currently clipped
PFQHIT_POINT	Intersection point in object coordinates

Table 6-3 (continued) Intersection-Query Token Names

Query Token	Description
PFQHIT_NORM	Geometric normal of an intersected triangle
PFQHIT_VERTS	Vertices of an intersected triangle
PFQHIT_TRI	Index of an intersected triangle
PFQHIT_PRIM	Index of an intersected primitive in pfGeoSet
PFQHIT_GSET	pfGeoSet of an intersection
PFQHIT_NODE	pfGeode of an intersection
PFQHIT_NAME	Name of pfGeode
PFQHIT_XFORM	Current transformation matrix
PFQHIT_PATH	Path in scene graph of intersection

The PFQHIT_FLAGS field is bit vector with bits that indicate whether an intersection occurred and whether the point, normal, primitive and transformation information is valid. For some types of intersections only some of the information has meaning; for instance, for a pfSegSet bounding volume intersecting a pfNode bounding sphere, the point information may not be valid.

Queries can be performed singly by calling **pfQueryHit()** with a single query token, or several at a time by using **pfMQueryHit()** with an array of tokens. In the latter case, the return information is placed in the specified order into a return array.

Intersection Masks

Before using **pfNodeIsectSegs()** to intersect the geometry in the scene graph, you must set intersection masks for the nodes in the scene graph and correspondingly in your search request.

Setting the Intersection Mask

pfNodeTravMask() sets the intersection masks in a subgraph of the scene down through GeoSets. For example:

```
pfNodeTravMask(root, PFTRAV_ISECT, 0x01,
               PFTRAV_SELF | PFTRAV_DESCEND, PF_SET)
```

sets the intersection mask of all nodes and GeoSets in the scene graph to 0x01. A subsequent intersection request would then use 0x01 as the mask in **pfNodeIsectSegs()**. A description of how to use this mask follows.

Specifying Different Classes of Geometry

Databases can contain different classes of objects, and only some of those may be relevant for a particular intersection request. For example, the wheels on a truck follow the ground, even through a small pond; therefore, you only want to test for intersection with the ground and not with the water. For a boat, on the other hand, intersections with both water and the lake bottom have significance.

To accommodate distinctions between classes of objects, each node and GeoSet in a scene graph has an intersection mask. This mask allows traversals, such as intersections, to either consider or ignore geometry by class.

For example, you could use four classes of geometry to control tests for collision detection of a moving ship, collision detection for a falling bowling ball, and line-of-sight visibility. Table 6-4 matches database classes with the pfNodeTravMask and pfGSetIsectMask values used to support the traversal tests listed above.

Table 6-4 Database Classes and Corresponding Node Masks

Database Class	Node Mask
Water	0x01
Ground	0x02
Pier	0x04
Clouds	0x08

Once the mask values at nodes in the database have been set, intersection traversals can be directed by them. For example, the line segments for ship collision detection should be sensitive to the water, ground, and pier, while a those for a bowling ball would ignore intersections with water and the clouds, testing only against the ground and pier. Line-of-sight ranging should be sensitive to all the geometry in the scene. Table 6-5 lists the traversal mask values and mask representations that would achieve the proper intersection tests.

Table 6-5 Representing Traversal Mask Values

Intersection Class	Mask Value	Mask Representation
Ship	0x07	(Water | Ground | Pier)
Bowling ball	0x06	(Ground | Pier)
Line-of-sight ranging	0x0f	(Water | Ground | Pier | Clouds)

The intersection traversal prunes a node as soon as it gets a zero result from doing a bitwise AND of the node intersection mask and the traversal mask specified by the pfSegSet's *isectMask* field. Thus, all nodes in the scene graph should normally be set to be the bitwise OR of the masks of their children. After setting the class-specific masks for different subgraphs of the scene, this can be accomplished by calling

```
pfNodeTravMask(root, PFSET_OR, PFTRAV_SET_FROM_CHILD, 0x0);
```

which sets each node's mask by OR-ing 0x0 with the current mask and the masks of the node's children.

Note that this traversal, like that used to update node bounding volumes, is unusual in that it propagates information up the graph from leaf nodes to root.

Discriminator Callbacks

If you need to make a more sophisticated discrimination than node masks allow about when an intersection is valid, IRIS Performer can issue a callback on each successful intersection and let you decide whether the intersection is valid in the current context.

If a callback is specified in **pfNodeIsectSegs()**, then at each level where an intersection occurs—for example, with bounding volumes of *libpf* pfGeodes (mode PFTRAV_IS_GEODE), *libpr* GeoSets (mode PFTRAV_IS_GSET), or individual geometric primitives (mode PFTRAV_IS_PRIM)—IRIS Performer invokes the callback, giving it information about the candidate intersection. The value you return from the callback determines whether the intersection should be ignored and how the intersection traversal should proceed.

If the return value includes the bit PFTRAV_IS_IGNORE, the intersection is ignored. The intersection traversal itself can also be influenced by the callback. The traversal is subject to three possible fates, as detailed in Table 6-6.

Table 6-6 Possible Traversal Results

Set Bits	Meaning
PFTRAV_CONT	Continue the traversal inside this subgraph or GeoSet.
PFTRAV_PRUNE	Continue the traversal but skip the rest of this subgraph or GeoSet.
PFTRAV_TERM	Terminate the traversal here.

Line Segment Clipping

Usually, the intersection point of most interest is the one that is nearest to the beginning of the segment. By default, after each successful intersection, the end of the segment is clipped so that the segment now ends at the intersection point. Upon the final return from the traversal, it contains the closest intersection point.

However, if you want to examine all intersections along a segment you can use a discriminator callback to tell IRIS Performer not to clip segments—simply leave out the PFTRAV_IS_CLIP_END bit in the return value. If you want the farthest intersection point, you can use PFTRAV_IS_CLIP_START so that after each intersection the new segment starts at the intersection point and extends outward.

Traversing Special Nodes

Level-of-detail nodes are intersected against the model for range zero, which is typically the highest level-of-detail. If you want to select a different model, you can turn off the intersection mask for the LOD node and place a switch node in parallel (having the same parent and children as the LOD) and set it to the desired model.

Sequences and switches intersect using the currently active child or children. Billboards are not intersected against, since no eyepoint is defined for intersection traversals.

Picking

pfChanPick() provides a simpler interface to intersection testing by using a mouse to select geometry. It uses **pfNodeIsectSegs()** and reserves the high bit, PFIS_PICK_MASK, of the intersection mask in the scene graph for its own use. Setting up picking with **pfNodePickSetup()** sets this bit in the intersection mask throughout the specified subgraph, but doesn't enable caching inside pfGeoSets (see the "Performance" section of this chapter).

pfChanPick() has an extra feature: It can either return the closest intersection (PFPK_M_NEAREST) or return all pfHits along the picking ray (PFPK_M_ALL).

Performance

The intersection traversal uses the hierarchical bounding volumes in the scene graph to allow culling of the database and then processes candidate GeoSets by testing against their internal geometry. For this reason, the hierarchy should reflect the spatial organization of the database. High-performance culling has similar requirements (see Chapter 13).

Performance Trade-offs

IRIS Performer currently retains no information about spatial organization of data within GeoSets, so each triangle in the GeoSet must be tested. Although large GeoSets are good for rendering performance in the absence

of culling, spatially localized GeoSets are best for culling (since a GeoSet is the smallest culling unit), and spatially localized GeoSets with few primitives are best for intersections.

Front Face/Back Face

One way to speed up intersection testing is to turn on PFTRAV_IS_CULL_BACK. When this flag is enabled, only front-facing geometry is tested.

Enabling Caching

Precomputing information about normals and projections speeds up intersections inside GeoSets. For the best performance, you should enable caching in GeoSets when you set the intersection masks with **pfNodeTravMask()**.

If the geometry within a GeoSet is dynamic, such as waves on a lake, caching can cause incorrect results. However, for geometry that changes only rarely, you can use **pfGSetIsectMask()** to recompute the cache as needed.

Intersection Methods for Segments

Normally, when intersecting down to the primitive level each line segment is separately tested against each bounding volume in the scene graph, and after passing those tests is intersected against the pfGeoSet bounding box. Segments that intersect the bounding box are eventually tested against actual geometry.

When a pfSegSet has a spatially localized group of at least several line segments, you can speed up the traversal by providing a bounding volume. You can use **pfCylAroundSegs()** to create a bounding cylinder for the segments, and place a pointer to the resulting cylinder in the pfSegSet's *bound* field; then OR the PFTRAV_IS_BCYL bit into the pfSegSet's *mode* field.

If only a rough volume-volume intersection is required, you can specify a bounding cylinder in the pfSegSet without any line segments at all and request discriminator callbacks at the PFTRAV_IS_NODE or PFTRAV_IS_GSET level.

Figure 6-4 illustrates some aspects of this process. The portion of the figure labeled A represents a single segment; B is a collection of nonparallel segments, not suitable for tightly bounding with a cylinder; and C shows parallel segments surrounded by a bounding cylinder. In the bottom portion of the figure, the bounding cylinder around the segments intersects the bounding box around the object; each segment in the cylinder thus must be tested individually to see if any of them intersect.

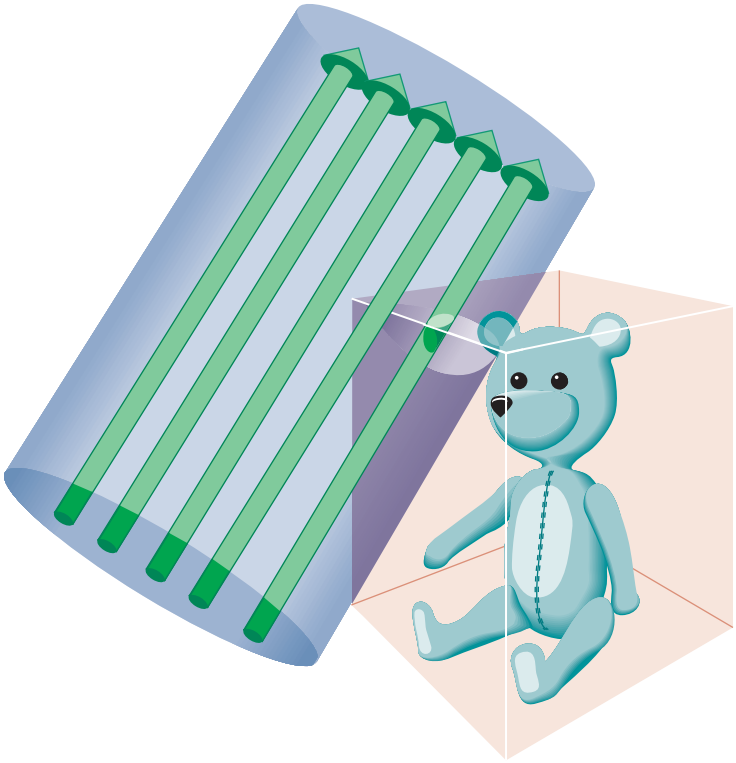


Figure 6-4 Intersection Methods

Chapter 7

“Frame and Load Control”

This chapter explains how to control frame rate, synchronization, and dynamic load management.

Chapter 7

Frame and Load Control

This chapter describes how to manage the display operations of a visual simulation application to maintain the desired frame rate and visual performance level.In addition this chapter covers advanced topics including multiprocessing and shared memory management.

Frame-Rate Management

A *frame* is the period of time in which all processing must be completed before updating the display with a new image, for example, a frame rate of 60Hz means the display is updated 60 times per second and the time extent of a frame is 16.7ms. The ability to fit all processing within a frame depends on several variables, some of which are:

- the number of pixels being filled
- the number of transformations and modal changes being made
- the amount of processing required to create a display list for a single frame
- the quantity of information being sent to the graphics subsystem

Through intelligent management of Silicon Graphics CPU and graphics hardware, IRIS Performer minimizes the above variables in order to achieve the desired frame rate. However, in some cases, peak frame rate is less important than a *fixed frame rate*. Fixed frame rate means that the display is updated at a consistent, unvarying rate. While a simple step towards achieving a fixed frame rate is to reduce the frame rate expectation, we shall explore other (less Draconian) mechanisms in this chapter that do not adversely impact frame rates.

As discussed in the following sections, IRIS Performer lets you select the frame rate and has built-in functionality to maintain that frame rate and control overload situations when the draw time exceeds or grows

uncomfortably close to a frame time. While these methods can be effective, they do require some cooperation from the run-time database. In particular, databases should be modeled with levels-of-detail and be spatially arranged.

Selecting the Frame Rate

IRIS Performer is designed to run at the fixed frame rate as specified by **pfFrameRate()**. Selecting a fixed frame rate does not in itself guarantee that each frame can be completed within the desired time. It is possible that some frames might require more computation time than is allotted by the frame rate. By taking too long, these frames cause *dropped* or *skipped* frames. A situation in which frames are dropped is called an *overload* or *overrun* situation. A system that is close to dropping frames is said to be in *stress*.

Achieving the Frame Rate

The first step towards achieving a frame rate is to make sure that the scene can be processed in less than a frame's time—hopefully *much* less than a frame's time. Although minimizing the processing time of a frame is a huge effort, rife with tricks and black magic, certain techniques stand out as IRIS Performer's main weapons against slothful performance:

- Multiprocessing. The use of multiple processes on multi-CPU systems can drastically increase throughput.
- View culling. By trivially rejecting portions of the database outside the viewing volume, performance can be increased by orders of magnitude.
- State sorting. Many graphics pipelines are sensitive to graphics mode changes. Sorting a scene by graphics state greatly reduces mode changes, increasing the efficiency of the hardware.
- Level-of-detail. Objects that are far away project to a relatively small area of the display so fewer polygons can be used to render the object without substantial loss of image quality. The overall result is fewer polygons to draw and improved performance.

Multiprocessing and level-of-detail is discussed in this chapter while view culling and state sorting are discussed in Chapter 6, “Database Traversal.” More information on sorting in the context of performance tuning can be found in Chapter 13, “Performance Tuning and Debugging.”

Fixing the Frame Rate

Frame intervals are fixed periods of time but frame processing is variable in nature. Because things change in a scene, such as when objects come into the field of view, frame processing cannot be fixed. In order to maintain a fixed frame rate, the average frame processing time must be less than the frame time so that fluctuations don’t exceed the selected frame time. Alternately, the scene complexity can be automatically reduced or increased so that the frame rate stays within a user-defined “sweet spot”. This mechanism requires that the scene be modeled with levels-of-detail (pfLOD nodes).

Each frame, IRIS Performer calculates the system *load* for each frame. Load is calculated as the percentage of the frame period it took to process the frame. Then if the default IRIS Performer fixed frame rate mechanisms are enabled, load is used to calculate system *stress*, which is in turn used to adjust the level of detail (LOD) of visible models. LOD management is IRIS Performer’s primary method of managing system load.

Table 7-1 shows the IRIS Performer functions for controlling frame processing.

Table 7-1 Frame Control Functions

Function	Description
pfFrameRate	Set the desired frame rate.
pfSync	Synchronize processing to frame boundaries.
pfFrame	Initiate frame processing.
pfPhase	Control frame boundaries.
pfChanStressFilter	Control how stress is applied to LOD ranges.
pfChanStress	Manually control the stress value.

Table 7-1 (continued) Frame Control Functions

Function	Description
pfGetChanLoad	Determine the current system load.
pfChanLODAttr	Control how LOD is performed, including global LOD adjustment and blending (fade).

Figure 7-1 shows a frame-timing diagram that illustrates what occurs when frame computations are not completed within the required interval. The solid vertical lines in Figure 7-1 represent frame-display intervals. The dashed vertical lines represent video refresh intervals.

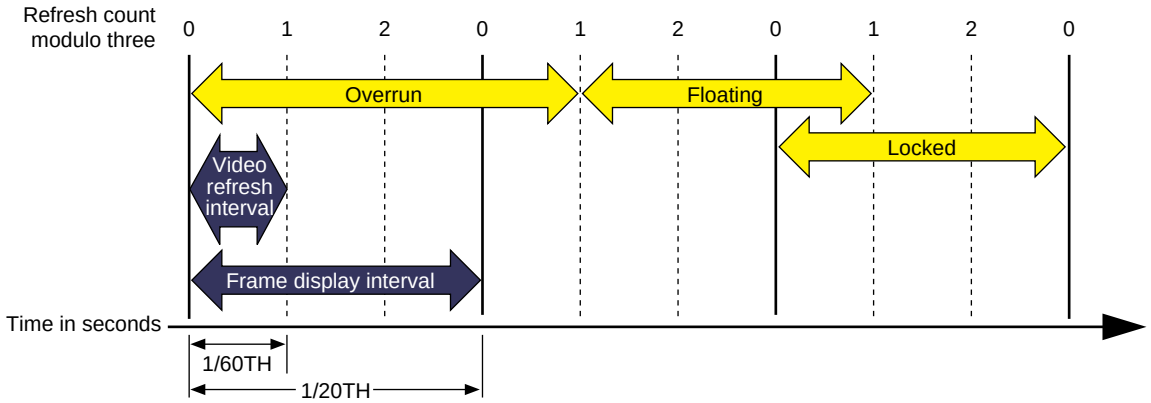


Figure 7-1 Frame Rate and Phase Control

In this example, the video scan rate is 60 Hz and the frame rate is 20 Hz. With the video hardware running at 60 Hz, each of the 20 Hz frames should be scanned to the video display three times, and the system should wait for every third vertical retrace signal before displaying the next image. The numbers across the top of the figure represent the refresh count modulo three. New images are displayed on refreshes whose count modulo three is zero, as shown by the solid lines.

In the first frame of this example, the new image isn't yet completed when the third vertical retrace signal occurs; the same image must therefore be displayed again during the next interval. This situation is known as frame *overrun*, because the frame computation time extends past a refresh boundary.

Frame Synchronization

Because of the overrun, the frame and refresh interval timing is no longer synchronized; it's out of phase. A decision must be made either to display the same image for the remaining two intervals, or to switch to the next image even though the refresh isn't aligned on a frame boundary. The frame-rate control mode, discussed in the next section, determines which choice is selected.

Knowing that the situation illustrated in Figure 7-1 is a possibility, you can specify a frame control mode to indicate what you would like the system to do when a frame overrun occurs.

To specify a method of frame-rate control, call **pfPhase()**. There are three available choices:

- Free run without phase control (PFPHASE_FREE_RUN) tells the application to run as fast as possible—to display each new frame as soon as it's ready, without attempting to maintain a constant frame rate.
- Free run without phase control but with a limit on the maximum frame rate (PFPHASE_LIMIT) tells the application to run no faster than a specified rate.
- Fixed frame rate with floating phase (PFPHASE_FLOAT) allows the drawing process to display a new frame (using **swapbuffers**(3G)) at any time, regardless of frame boundaries.
- Fixed frame rate with locked phase (PFPHASE_LOCK) requires the draw process to wait for a frame boundary before displaying a new frame.

Free-Running Frame-Rate Control

The simplest form of frame-rate control, called *free-running*, is to have no control at all. This uncontrolled mode draws frames as quickly as the

hardware is able to process them. In free-running mode, the frame rate may be 60 Hz in the areas of low database complexity, but could drop to a slower rate in views that place greater demand on the system. Use **pfPhase(PFPHASE_FREE_RUN)** to specify a free-running frame rate.

In applications in which real-time graphics provide the majority of visual cues to an observer, the variable frame rates produced by the free-running mode may be undesirable. The variable lag in image update associated with variable frame rate can lead to motion sickness for the simulation participants, especially in motion platform-based trainers or immersive head-mounted displays. For these and other reasons it is usually preferable to maintain a steady, consistent frame-update rate.

Fixed Frame-Rate Control

Assume that the overrun frame in Figure 7-1 completes processing during the next refresh period, as shown. After the overrun frame, the simulation is still running at the chosen 20-Hz rate and is updating at every third vertical retrace. If a new image is displayed at the next refresh, its start time lags by 1/60th of a second, and therefore it is out of phase by that much.

Subsequent images are displayed when the refresh count modulo three is one. As the simulation continues and additional extended frames occur, the phase continues to drift. This mode of operation is called *floating phase*, as shown by the frame in Figure 7-1 labeled *Floating*. Use **pfPhase(PFPHASE_FLOAT)** to select floating-phase frame control.

The alternative to displaying a new image out of phase is to display the old image for the remainder of the current update period, then change to the new image at the normal time. This *locked phase* extends each frame overrun to an integral multiple of the selected frame time, making the overrun more evident but also maintaining phase throughout the simulation. This timing is shown by the frame in Figure 7-1 labeled *Locked*. Although this mode is the most restrictive, it is also the most desirable in many cases. Use **pfPhase(PFPHASE_LOCK)** to select phase-locked frame control.

For example, a 20-Hz phase-locked frame rate is selected by specifying:

```
pfPhase(PFPHASE_LOCK);
pfFrameRate(20.0f);
```

These specifications prevent the system from switching to a newly computed image until a display period of 1/20th second has passed from the time the previous image was displayed. The frame rate remains fixed even when the Geometry Pipeline finishes its work in less time. Fixed frame-rate display therefore involves setting the desired frame rate and selecting one of the two fixed-frame-rate control modes.

Frame Skipping

When multiple frame times elapse during the rendering of a single frame, the system must choose which frame to draw next. If the per-frame display lists are processed in strict succession even after a frame overrun, the visual image slowly recedes in time and the positional correlation between display and simulation is lost. To avoid this problem, only the most recent frame definition received by the draw process is sent to the Geometry Pipeline, and all intervening frame definitions are abandoned. This is known as *dropping* or *skipping* frames and is performed in both of the fixed frame-rate modes.

Because the effects of variable frame rates, phase variance, and frame dropping are distracting, you should choose a frame rate with care. Steady frame rates are achieved when the frame time allows the worst-case view to be computed without overload. The structure of the visual database, particularly in terms of uniform “complexity density,” can be important in maximizing the system frame rate. See “Organizing a Database for Efficient Culling” in Chapter 6 and Figure 6-3 for examples of the importance of database structure.

Maintaining a fixed frame rate involves managing future system load by adjusting graphics display actions to compensate for varying past and present loads. The theory behind load management and suggested methods for dealing with variable load situations are discussed in the “Level-of-Detail Management” section of this chapter.

Sample Code

Example 7-1 demonstrates a common approach to frame control. The code is based on part of the *main.c* source file used in the *perfly* sample application.

Example 7-1 Frame Control Excerpt

```
/* Set the desired frame rate. */
pfFrameRate(ViewState->frameRate);

/* Set the MP synchronization phase. */
pfPhase(ViewState->phase);

/* Application main loop */
while (!SimDone())
{
    /* Sleep until next frame */
    pfSync();

    /* Should do all latency-critical processing between
     * pfSync() and pfFrame().  Such processing usually
     * involves changing the viewing position.
     */
    PreFrame();

    /* Trigger cull and draw processing for this frame. */
    pfFrame();

    /* Perform non-latency-critical simulation updates. */
    PostFrame();
}
```

Level-of-Detail Management

All graphics systems have finite capabilities that affect the number of geometric primitives that can be displayed per frame at a specified frame rate. Because of these limitations, maximizing visual cues while minimizing the polygon count in a database is often an important aspect of database development. Level-of-detail processing is one of the most beneficial tools available for managing database complexity for the purpose of improving display performance.

The basic premise of LOD processing is that objects that are barely visible, either because they are located a great distance from the eyepoint or because atmospheric conditions reduce visibility, don't need to be rendered in great detail in order to be recognizable. This is in stark contrast to brutishly mandating that all polygons be rendered regardless of their contribution to the visual scene. Both atmospheric effects and the visual effect of perspective decrease the importance of details as range from the eyepoint increases. The predominant visual effect of distance is the perspective foreshortening of objects, which makes them appear to shrink in size as they recede into the distance.

To save rendering time, objects that are visually less important in a frame can be rendered with less detail. The LOD approach to optimizing the display of complex objects is to construct a number of progressively simpler versions of an object and to select one of them for display as a function of range.

This requires you to create multiple models of an object with varying levels of detail. You also must supply a rule to determine how much detail is appropriate for a given distance to the eyepoint. The sections that follow describe how to create multiple LOD models and how to control when the changeover to a different LOD occurs.

Level-of-Detail Models

Most objects comprise smaller objects that become visually insignificant at ranges where the conglomerate object itself is still quite prominent. For example, a complex model of an automobile might have door handles, side- and rear-view mirrors, license plates, and other small details.

A short distance away, these features may no longer be visible, even though the car itself is still a visually significant element of the scene. It is important to realize that as a group, these small features may contain as many polygons as the larger car itself, and thus have a detrimental effect on rendering speed.

You can construct two LOD models simply by providing one model that contains all of the detailed features and another model that contains only the car body itself and none of the detailed features. A more sophisticated scheme uses multiple LOD models that are grouped under an LOD node.

Figure 7-2 shows an LOD node with multiple children numbered 1 through *n*. In this case, the model named LOD 1 is the most detailed model and models LOD 2 through LOD *n* represent progressively coarser models. Each of these LOD models might contain children that also have LOD components. Associated with the LOD node is a list of ranges that define the distance at which each model is appropriate to display. There is no limit to the number of levels of detail that can be used.

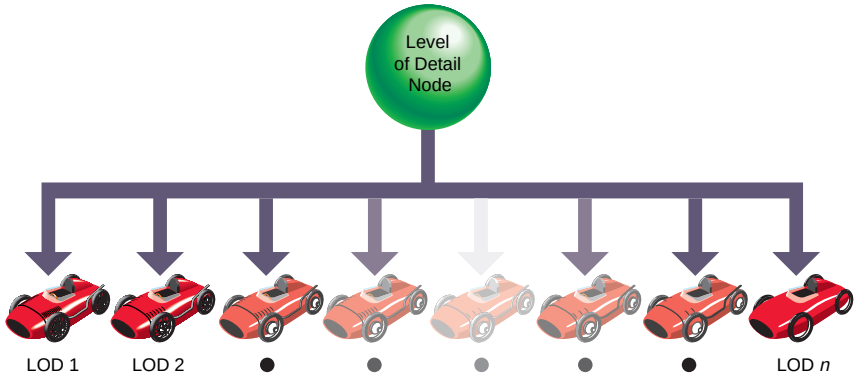


Figure 7-2 Level-of-Detail Node Structure

The object can be transformed as needed. During the culling phase of frame processing, the distance from the eyepoint to the object is computed and used (with other factors) to select which LOD model to display.

The IRIS Performer pfLOD node contains a value known as the center of LOD processing. The LOD center point is an *x*, *y*, *z* location that defines the point used in conjunction with the eyepoint for LOD range-switching calculations, as described in the “Level-of-Detail Range Processing” section of this chapter.

Figure 7-3 shows an example in which multiple LOD models grouped under a parent LOD node are used to represent a toy racecar.

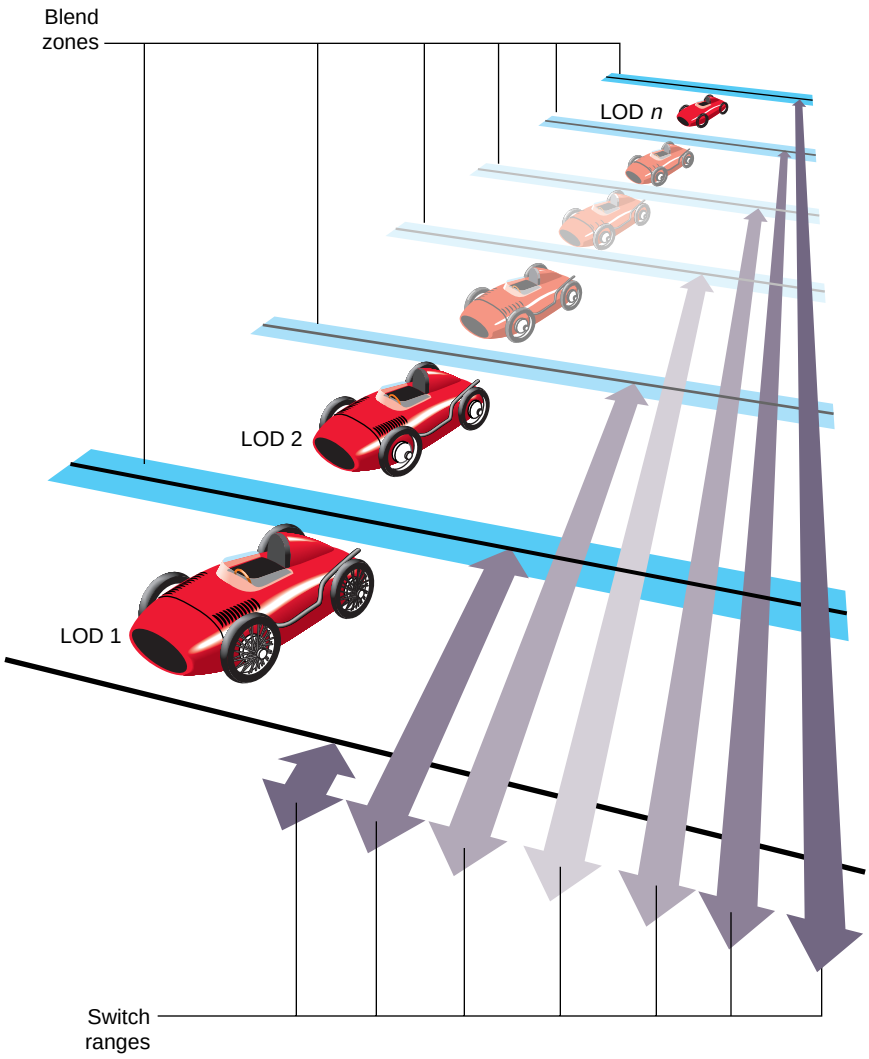


Figure 7-3 Level-of-Detail Processing

Figure 7-3 demonstrates that each car in a row of identical cars placed at increasing range from the eyepoint is drawn using a different child of the tree's LOD node.

The double-ended arrows indicate a switch range for each level of detail. When the car is closer to the eyepoint than the first range, nothing is drawn. When the car is between the first and second ranges, LOD 1 is drawn. When the car is between the second and third ranges, LOD 2 is drawn.

This range bracketing continues until the final range is passed, at which point nothing is drawn. The pfLOD node's switch range list contains one more entry than the number of child nodes to allow for this range bracketing.

IRIS Performer provides the ability to specify a blend zone for each switch between LOD models. Such that pfLOD nodes now also hold a list of these transition distances over which Performer should 'blend' between neighboring LODs. These blend zones will be discussed in more detail in "Level-of-Detail Transition Blending" on page 207.

Level of Detail States

IRIS Performer 2.0 supports a new concept along with the aforementioned standard LOD node which contains switch ranges and transition zones. This new concept is one of an LOD state—the pfLODState. A pfLODState is an essence a way of creating classes or priorities among LODs. A pfLODstate contains eight parameters used to modify four different ways in which Performer calculates LOD switch ranges and LOD transition distances. LOD states contain the following parameters:

- Scale for LODs switch Ranges.
- Offset for LODs switch Ranges.
- Scale for the effect of Stress of switch Ranges.
- Offset for the effect of Stress on switch Ranges.
- Scale for the transition distances per LOD switch
- Offset for the transition distances per LOD switch
- Scale for the effect of Stress on transition distances
- Offset for the effect of Stress on transition distances

These LOD states can then be attached to either single or multiple LOD nodes such that the LOD behavior of groups or classes of objects can be different and be easily modified. The reference pages for **pfLODLODState()** and **pfLODLODStateIndex()** contain detailed information on how to attach pfLODStates.

LOD states are useful because in a particular scene there often exists an object of focus such as a sign, a target, or some other object of particular visual significance that needs to be treated specially with regard to visual importance and thus LOD behavior. It stands to reason, that this particular object (or small group of objects) should be at the highest detail possible despite being farther away than other elements in the scene which might not be as visually significant. In fact, it might be feasible to diminish the detail of these less important objects (like rocks and trees) in favor of the other more important objects (despite these objects being further relatively in range). In this case one would just create two LOD states. The first would be for the important objects and would effectively disable the effect of stress on these nodes as well as scale the switch ranges such that the object(s) would maintain more detail for further ranges. The second LOD state would be used to make the objects of less importance be more responsive to system stress and possibly scale their switch ranges such that they would show even less detail than normal. In this way, LOD states allow biasing among different LODs to maintain desirable rendering speeds while maintaining the visual integrity of various objects depending on their subjective importance (rather than solely on their current visual significance).

In some multichannel applications, LOD states are used to control the action of LODs in different viewing channels that have different visual significance criteria—for instance one channel might be a normal channel while a second might represent an infra-red display. Rather than simple use of LOD states, it is also possible to specify a list of LOD states to a channel and use indexes from this list for particular LODs (via **pfChanLODStateList()** and **pfLODLODStateIndex()**). In this way, in the normal channel a car's geometry might be particularly important while in the infra-red channel, the hot exhaust of the same car might be much more important to observe. This type of channel dependent LOD can be set up by using two distinct and different LOD states for the same index in the lists of LOD states specified for unique channels.

Note that because IRIS Performer performs LOD calculations in a range squared space as much as possible for efficiency reasons, LOD computation becomes more costly when LOD states contain scales that are not equal to 1.0 or offsets not equal to 0.0 for transitions or switch ranges—these offsets force IRIS Performer to perform otherwise avoidable square roots calculations in order to correctly calculate the effects of scale and offset on the LOD.

Level-of-Detail Range Processing

The LOD switch ranges present in LOD nodes are processed before being used to make the level of detail selection. The goal of range setting is to switch LODs as objects reach certain levels of perceptibility. The size of a channel in pixels, the field of view used in viewing, and the distance from the observer to the display surface all affect object perceptibility.

IRIS Performer uses a channel size of 1024x1024 pixels and a 45-degree field of view as the basis for calculating LOD switching ranges. The screen space size of a channel and the current field of view are used to compute an LOD scale factor that is updated whenever the channel size or the field of view changes.

There is an additional global LOD scale factor that can be used to adjust switch ranges based on the relationship between the observer and the display surface. The default global scale factor is 1.

Note that LOD switch ranges are also effected by LOD states that have been attached to either a particular LOD or to a channel that contains the LOD. These LOD states provide the mechanism to apply both a scale and an offset for an LODs switch ranges and to the effect of system stress on those switch ranges. See “Level of Detail States” on page 202 for more information of pfLODStates.

Ultimately a LODs switch range without regard to system stress can be computed as follows:

```
switch_range[i] =
    (range[i] *
        LODStateRangeScale *
        ChannelLODStateRangeScale +
        LODStateRangeOffset +
        ChannelLODStateRangeOffset) *
```

```
        ChannelLODScale *
        ChannelSizeAndFOVFactor;
```

If IRIS Performer channel stress processing is active, the computed range is modified as follows:

```
    switch_range[i] *=
        (ChannelLODStress *
        LODStateRangeStressScale *
        ChannelLODStateRangeStressScale +
        LODStateRangeStressOffset +
        ChannelLODStateRangeStressOffset);
```

Example 7-2 illustrates how to set LOD ranges. The code in this example is derived from */usr/share/Performer/src/sample/apps/C/lod/lod.c*, the main source module for the sample program *lod*.

Example 7-2 Setting LOD Ranges

```
/* setLODRanges() -- sets the ranges for the LOD node. The
 * ranges from 0 to NumLODs are equally spaced between min
 * and max.  The last range, which determines how far you
 * can get from the object and still see it, is set to
 * visMax.
 */
void
setLODRanges(pfLOD *lod,float min, float max, float visMax)
{
    int i;
    float range, rangeInc;

    rangeInc = (max - min)/(ViewState->shellLOD + 1);
    for (range = min, i = 0; i < ViewState->shellLOD; i++)
    {
        ViewState->range[i] = range;
        pfLODRange(lod, i, range);
        range += rangeInc;
    }
    ViewState->range[i] = visMax;
    pfLODRange(lod, i, visMax);
}

/* generateShellLODs() -- creates shell LOD nodes according
 * to the parameters specified in the shared data structure.
 */
```

```
void
generateShellLODs(void)
{
    int i;
    pfGroup *grp;
    pfVec4 clr;
    long numLOD = ViewState->shellLOD;
    long numPnts = ViewState->shellPnts;
    long numPcs = ViewState->shellPcs;

    for (i = 1; i <= numLOD; i++)
    {
        if (ViewState->shellColor == SHELL_COLOR_SING)
            pfSetVec4(clr, 0.9f, 0.1f, 0.1f, 1.0f);
        else
            /* set the color.  highest level = RED;
             * middle LOD = GREEN; lowest LOD = BLUE
             */
            pfSetVec4(clr,
                  (i <= (long)floor((double)(numLOD/2.0f)))?
                     (-2.0f/numLOD) * i + 1.0f + 2.0f/numLOD:
                     0.0f,
                  (i <= (long)floor((double)(numLOD/2)))?
                     (2.0f/numLOD) * (i - 1):
                     (-2.0f/numLOD) * i + 2.0f,
                  (i <= (long)floor((double)(numLOD/2)))?
                     0.0f:
                     (2.0f/numLOD) * i - 1.0f,
                  1.0f);

        /* build a shell GeoSet */
        grp = createShell(numPcs, numPnts,
                          ViewState->shellSweep, &clr,
                          ViewState->shellDraw);
        normalizeNode((pfNode *)grp);

        /* add geode as another level of detail node */
        pfAddChild(ViewState->LOD, grp);

        /* simplify the geometry, but don't have less than
         * 4 points per circle or less than 3 pieces */
        numPnts = (numPnts > 7) ? numPnts-4 : 4;
        numPcs = (numPcs > 6) ? numPcs-4 : 3;
    }
}
```

```
...
    ViewState->LOD = pfNewLOD();

    generateShellLODs();

    /* get the LOD's extents */
    pfGetNodeBSphere(ViewState->LOD, &(ViewState->bSphere));
    pfLODCenter(ViewState->LOD, ViewState->bSphere.center);
    /* set ranges for LODs; there should be (num LODs + 1)
     * range entries */
    setLODRanges(ViewState->LOD, ViewState->minRange,
                 ViewState->maxRange, ViewState->max);
```

Level-of-Detail Transition Blending

An undesirable effect called *popping* occurs when the sudden transition from one LOD to the next LOD is visually noticeable. This distracting image artifact can be ameliorated with a slight modification to the normal LOD-switching process.

In this modified method a transition per LOD switch is established rather than making a sudden substitution of models at the indicated switch range. These transitions specify distances over which to blend between the previous and next LOD. These zones are considered to be centered at the specified LOD switch distance, as shown by the horizontal shaded bars of Figure 7-3. Note that Performer limits the transition distances to be equal to the shortest distance between the switch range and the two neighboring switch ranges. For more information, see the **pfLODTransition()** reference page.

As the range from eyepoint to LOD center-point transitions the blend zone, each of the neighboring LOD levels is drawn by using transparency to composite samples taken from the present LOD model with samples taken from the next LOD model. For example, at the near, center, and far points of the transition blend zone between LOD 1 and LOD 2, samples from both LOD 1 and LOD 2 are composited until the end of the transition zone is reached, where all the samples are obtained from LOD 2.

Table 7-2 lists the transparency factors used for transitioning from one LOD range to another LOD range.

Table 7-2 LOD Transition Zones

Distance	LOD 1	LOD 2
Near edge of blend zone	100% opaque	0% opaque
Center of blend zone	50% opaque	50% opaque
Far edge of blend zone	0% opaque	100% opaque

LOD transitions are made smoother and much less noticeable by applying a blending technique rather than making a sudden transition. Blending allows LOD transitions to look good at ranges closer to the eye than LOD popping allows. Decreasing switch ranges in this way improves the ability of LOD processing to maximize the visual impact of each polygon in the scene without creating distracting visual artifacts.

The benefits of smooth LOD transition have an associated cost. The expense lies in the fact that when an object is within a blend zone, two versions of that object are drawn. This causes blended LOD transitions to increase the scene polygon complexity during the time of transition. For this reason, the blend zone is best kept to the shortest distance that avoids distracting LOD-popping artifacts. Currently, fade level of detail is supported only on RealityEngine™ graphics systems.

Note that the actual ‘blend’ or ‘fade’ distance used by Performer can also be adjusted by the LOD priority structures called pfLODStates. pfLODStates hold an offset and scale for the size of transition zones as well as an offset and scale for how system stress can affect the size of the transition zones. See “Level of Detail States” on page 202 for more information on pfLODStates.

Note also, that there exists a global LOD transition scale on a per channel basis that can affect all transition distances uniformly.

Thus for an LOD with 5 switch ranges R0, R1, R2, R3, R4 to switch between four models (M0, M1, M2, M3), there are 5 transition zones T0 (fade in M0), T1 (blend between M0 and M1), T2 (blend between M1 and M2), T3 (blend between M2 and M3), T4 (fade out M3). The actual fade distances (without regard to channel stress) are as follows.

```
fadeDistance[i] =
    (transition[i] *
        LODStateTransitionScale *
        ChannelLODStateTransitionScale +
        LODStateTransitionOffset +
        ChannelLODStateTransitionOffset) *
    ChannelLODFadeScale;
```

If Performer management of channel stress is turned on then the above fade distance is modified as follows:

```
fadeDistance[i] /=
    (ChannelStress *
    LODStateTransitionStressScale *
    ChannelLODStateTransitionStressScale +
    LODStateTransitionStressOffset +
    ChannelLODStateTransitionStressOffset);
```

Terrain Level of Detail

In creating LOD models and transitions for objects, it's often safe to assume that the *entire* model should transition at the same time. It's quite reasonable to make features of an automobile such as door handles disappear from the scene at the same time even when the passenger door is slightly closer than the driver's door. It is much less clear that this approach would work for very large objects such as an aircraft carrier or a space station, and it's clearly not acceptable for objects that span a large extent, such as a terrain surface.

Attempts to handle large-extent objects with discrete LOD tools focus on breaking the big object into myriad small objects and treating each small object independently. This works in some cases but often fails at the junction between two or more independent objects where cracks or seams exist when different detail levels apply to the objects. Some terrain processing systems have attempted to provide a hierarchy of crack filling geometry that is enabled based on the LOD selections of two neighboring terrain patches. This "digital grout" becomes untenable when more than a few patches share a common vertex.

An alternate approach—the active surface definition—treats the entire terrain as a single connected (and coherent) surface. The essence of the approach is that a triangle representing terrain either is judged to be

sufficiently accurate or is recursively subdivided. The general manner of subdivision is to introduce one or more vertices along the sides of the original triangle. Connecting these new vertices to make triangles leads to the subdivision of the original triangle into multiple triangles. After subdivision, the algorithm perturbs the vertex positions. If the resulting triangles are not yet sufficiently fine, the subdivision process is applied to each one in turn based on range and other fidelity criteria.

Arbitrary Morphing

Terrain level of detail using an interpolative active surface definition is a restricted form of the more general notion of object morphing. Morphing of models such as the car in a previous example can simply involve scaling a small detail to a single point and then removing it from the scene. Morphing is possible even when the topologies of neighboring pairs do not match. Both models and terrain can have vertex, normal, color, and appearance information interpolated between two or more representations. The advantages of this approach include: reduced graphics complexity since blending is not used, constant intersection truth for collision and similar tasks, and monotonic database complexity that makes system load management much simpler.

Dynamic Load Management

Because the effects of variable image update rates can be objectionable, many simulation applications are designed to operate at a fixed frame rate. One approach to selecting this fixed frame rate is to select an update rate constrained by the most complex portion of the visual database. Although this conservative approach may be acceptable in some cases, IRIS Performer supports a more sophisticated approach using dynamic LOD scaling.

Using multiple LOD models throughout a database provides the traversal system with a parameter that can be used to control the polygonal complexity of models in the scene. The complexity of database objects can be reduced or increased by adjusting a global LOD range multiplier that determines which LOD level is drawn.

Using this facility, a closed-loop control system can be constructed that adjusts the LOD-switching criteria based on the system load, also called *stress*, in order to maintain a selected frame rate.

Figure 7-4 illustrates a stress-processing control system.

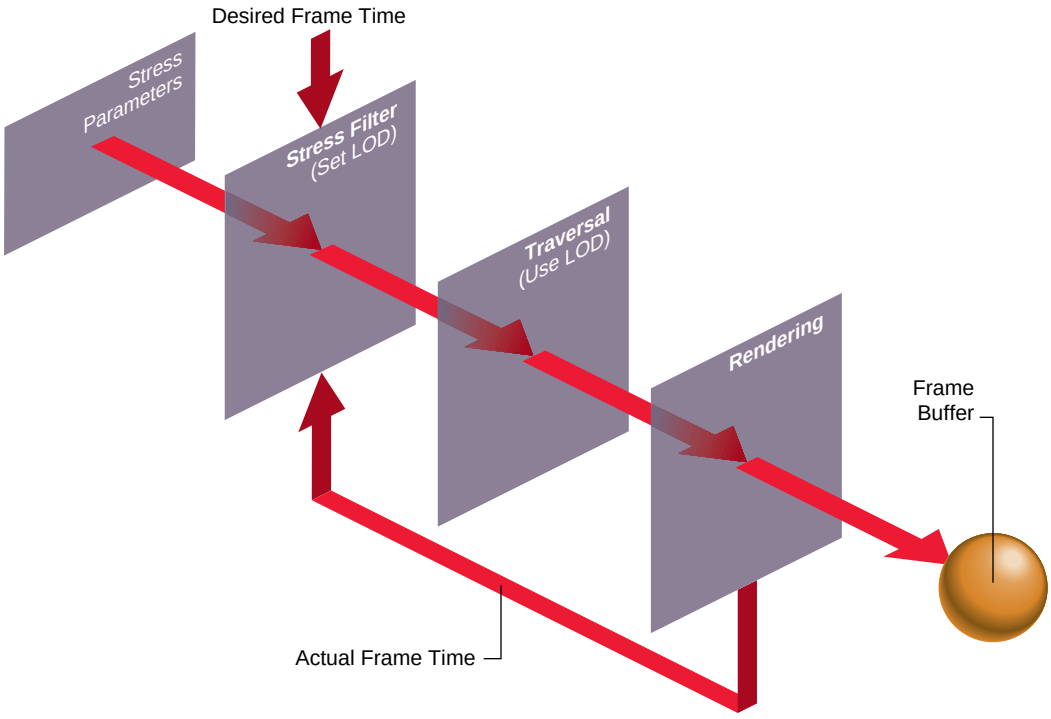


Figure 7-4 Stress Processing

In Figure 7-4, the desired and actual frame times are compared by the stress filter. Based on the user-supplied stress parameters, the stress filter adjusts the global LOD scale factor by increasing it when the system is overloaded and decreasing it when the system is underloaded. In this way, the system load is monitored and adjusted before each frame is generated.

The degree of stability for the closed-loop control system is an important issue. The ideal situation is to have a *critically damped* control system—that is, one in which just the right amount of control is supplied to maintain the

frame rate without introducing undesirable effects. The effects of overdamped and underdamped systems are visually distracting. An underdamped system oscillates, causing the system to continuously alternate between two different LOD models without reaching equilibrium. Overdamped systems may fail to react within the time required to maintain the desired frame rate. In practice, though, dynamic load management works well, and simple stress functions can handle the slowly changing loads presented by many databases.

The default stress function is controlled with user-selectable parameters. These parameters are set using the function **pfChanStressFilter()**.

The default stress function is implemented by the code fragment in Example 7-3.

Example 7-3 Default Stress Function

```
/* current load */
curLoad = drawTime * frameRate * frameFrac;

/* integrated over time */
if (curLoad < lowLoad)
    stressLevel -= stressParam * stressLevel;
else
if (curLoad > highLoad)
    stressLevel += stressParam * stressLevel;

/* limited to desired range */
if (stressLevel < 1.0)
    stressLevel = 1.0;
else
if (stressLevel > maxStress)
    stressLevel = maxStress;
```

The parameters *lowLoad* and *highLoad* define a “comfort zone” for the control system. The first if-test in the code fragment demonstrates that this comfort zone acts as a dead band. Instantaneous system load within the bounds of the dead band doesn’t result in a change in the system stress level. If the size of the comfort zone is too small, oscillatory distress is the probable result. It is often necessary to keep the *highLoad* level below the 100% point so that blended LOD transitions don’t drive the system into overload situations.

For those applications in which the default stress function is either inappropriate or insufficient, you can compute the system stress yourself and then set the stress load factor. Your filter function can access the same system measures that the default stress function uses, but it's also free to keep historical data and perform any feedback-transfer processing that application-specific dynamic load management may require.

The primary limitation of the default stress function is that it has a reactive rather than predictive nature. One of the major advantages of user-written stress filters is their ability to predict future stress levels before increased or decreased load situations reach the pipeline. Often the simulation application knows, for example, when a large number of moving models will soon enter the viewing frustum. If their presence is anticipated, then stress can be artificially increased so that no sudden LOD changes are required as they actually enter the field of view.

Successful Multiprocessing With IRIS Performer

This section describes an advanced topic that applies only to systems with more than one CPU. If you don't have a multiple-CPU system, you may want to skip this section.

IRIS Performer uses multiprocessing to increase throughput for both rendering and intersection detection. Multiprocessing can also be used for tasks that run asynchronously from the main application like database management. Although IRIS Performer hides much of the complexity involved, you need to know something about how multiprocessing works in order to use multiple processors well.

Review of Rendering Stages

Recall from Chapter 3 that an IRIS Performer application renders images using one or more pfPipes as independent software-rendering pipelines. The flow through the rendering pipeline can be modeled using these functional *stages*:

Intersection	Test for intersections between segments and geometry to simulate collision detection or line-of-sight for example.
Application	Do requisite processing for the visual simulation application, including reading input from control devices, simulating the vehicle dynamics of moving models, updating the visual database, and interacting with other networked simulation stations.
Cull	Traverse the visual database and determine which portions of it are potentially visible, perform level-of-detail selection for models with multiple representations, and build sorted, optimized display list for the draw stage.
Draw	Issue graphics library commands to a Geometry Pipeline in order to create an image for subsequent display.

You can partition these stages into separate parallel processes in order to distribute the work among multiple CPUs. Depending on your system type and configuration, you can use any of several available multiprocessing models.

Choosing a Multiprocessing Model

Use **pfMultiprocess()** to specify which functional stages, if any, should be forked into separate processes. The multiprocessing mode is actually a bitmask where each bit indicates that a particular stage should be configured as a separate process. For example, the bit PFMP_FORK_DRAW means the

draw stage should be split into its own process. Table 7-3 lists some convenience tokens that represent common multiprocessing modes:

Table 7-3 Multiprocessing Models

Model Name	Description
PFMP_APPCULLDRAW	Combine the application, cull, and draw stages into a single process. In this model, all of the stages execute within a single frame period. This is the minimum-*latency* mode of operation.
PFMP_APP_CULLDRAW or PFMP_FORK_CULL	Combine the cull and draw stages in a process that is separate from the application process. This model provides a full frame period for the application process, while culling and drawing share this same interval. This mode is appropriate when the host's simulation tasks are extensive but graphic demands are light, as might be the case when complex vehicle dynamics are performed but only a simple dashboard gauge is drawn to indicate the results.
PFMP_APPCULL_DRAW or PFMP_FORK_DRAW	Combine the application and cull stages in a process that is separate from the draw process. This mode is appropriate for many simulation applications when application and culling demands are light. It allocates a full CPU for drawing and has the APP and CULL stages share a frame period. Like the PFMP_APP_CULLDRAW mode, this mode has a single frame period of pre-draw latency.
PFMP_APP_CULL_DRAW or PFMP_FORK_CULL | PFMP_FORK_DRAW	Perform the application, cull, and draw stages as separate processes. This is the full maximum-throughput multiprocessing mode of IRIS Performer operation. In this mode, each pipeline stage is allotted a full frame period for its processing. Two frame periods of latency exist when using this high degree of parallelism.

You can also use **pfMultiprocess()** to specify the method of communication between the cull and draw stages, using the bitmasks PFMP_CULLoDRAW and PFMP_CULL_DL_DRAW.

Cull-Overlap-Draw Mode

Setting PFMP_CULLoDRAW specifies that the cull and draw processes for a given frame should overlap—that is, that they should run concurrently. For this to work, the cull and draw stages must be separate processes (PFMP_FORK_DRAW must be true). In this mode the two stages communicate in the classic producer-consumer model, by way of a pfDispList that is configured as a ring (FIFO) buffer; the cull process puts commands on the ring while the draw process simultaneously consumes these commands.

The main benefit of using PFMP_CULLoDRAW is reduced latency, since the number of pipeline stages is reduced by one and the resulting latency is reduced by an entire frame time. The main drawback is that the draw process must wait for the cull process to begin filling the ring buffer.

Forcing Display-List Generation

When the cull and draw stages are in separate processes, they communicate through a pfDispList; the cull process generates the display list, and the draw process traverses and renders it. (The display list is configured as a ring buffer when using PFMP_CULLoDRAW mode, as described in the “Cull-Overlap-Draw Mode” section).

However, when the cull and draw stages are in the same process (as occurs with the PFMP_APPCULLDRAW or PFMP_APP_CULLDRAW multiprocessing models) a display list isn’t required and by default one will not be used. Leaving out the pfDispList eliminates overhead. When no display list is used, the cull trigger function **pfCull()** has no effect; the cull traversal takes place when the draw trigger function **pfDraw()** is invoked.

In some cases you may want an intermediate pfDispList between the cull and draw stages even though those stages are in the same process. The most common situation that calls for such a setup is multipass rendering, when you want to cull only once but render multiple times. With PFMP_CULL_DL_DRAW enabled, **pfCull()** generates a pfDispList that can be rendered multiple times by multiple calls to **pfDraw()**.

Intersection Pipeline

The *intersection pipeline* is a two-stage pipeline consisting of the application and the intersection stages. The intersection stage may be configured as a separate process by setting the PFMP_FORK_ISECT bit in the bitmask given to **pfMultiprocess()**. When configured as such, the intersection process is triggered for the current frame when the application process calls **pfFrame()**. Then in the special intersection callback set with **pfIsectFunc()**, you can invoke any number of intersection requests with **pfNodeIsectSegs()**.

The intersection process is asynchronous so that if it does not finish within a frame time it does not slow down the rendering pipeline(s).

Multiple Pipelines

By default, IRIS Performer uses a single pfPipe, which in turn draws one or more pfChannels into one or more pfPipeWindows. If you want to use multiple rendering pipelines, as on two- or three-Geometry Pipeline Onyx RealityEngine^2 systems, use **pfMultipipe()** to specify the number of pfPipes required. When using multiple pipelines, the PFMP_APPCULLDRAW and PFMP_APPCULL_DRAW modes are not supported and IRIS Performer defaults to the PFMP_APP_CULL_DRAW multiprocessing configuration. Regardless of the number of pfPipes, there is always a single application process which triggers the rendering of all pipes with **pfFrame()**.

Multithreading

For additional multiprocessing and attendant increased throughput, the CULL stage of the rendering pipeline may be *multithreaded*. Multithreading means that a single pipeline stage is split into multiple processes, or threads which concurrently work on the same frame. Use **pfMultithread()** to allocate a number of threads for the cull stage of a particular rendering pipeline.

Cull multithreading takes place on a per-pfChannel basis, that is, each thread does all the culling work for a given pfChannel. Thus, an application with only a single channel will not benefit from multithreading the cull and an application with multiple, equally complex channels will benefit most by allocating a number of cull threads equal to the number of channels.

However, it is legal to allocate fewer cull threads if you do not have enough CPUs—in this case the threads are assigned to channels on a need basis.

Order of Calls

The multiprocessing model set by **pfMultiprocess()** is used for each of the rendering pipelines. In programs that configures the application stage as a separate process, all IRIS Performer calls must be made from the process that calls **pfConfig()** or the results are undefined. Both **pfMultiprocess()**, **pfMultithread()**, and **pfMultipipe()** must be called after **pfInit()** but before **pfConfig()**. **pfConfig()** configures IRIS Performer according to the required number of pipelines and the desired multiprocessing and multithreading modes, forks the appropriate number of processes, and then returns control to the application. **pfConfig()** should be called only once during each IRIS Performer application.

Comparative Structure of Models

Figure 7-5 shows timing diagrams for each of the process models. The vertical lines are frame boundaries. Five frames of the simulation are shown to allow the system to reach steady-state operation.

Boxes represent the functional stages and are labeled as follows:

A_n	application process for the *n*th frame
C_n	cull process for the *n*th frame
D_n	draw process for the *n*th frame

Only one of these models can be selected at a time, but they are shown together so that you can compare their structures.

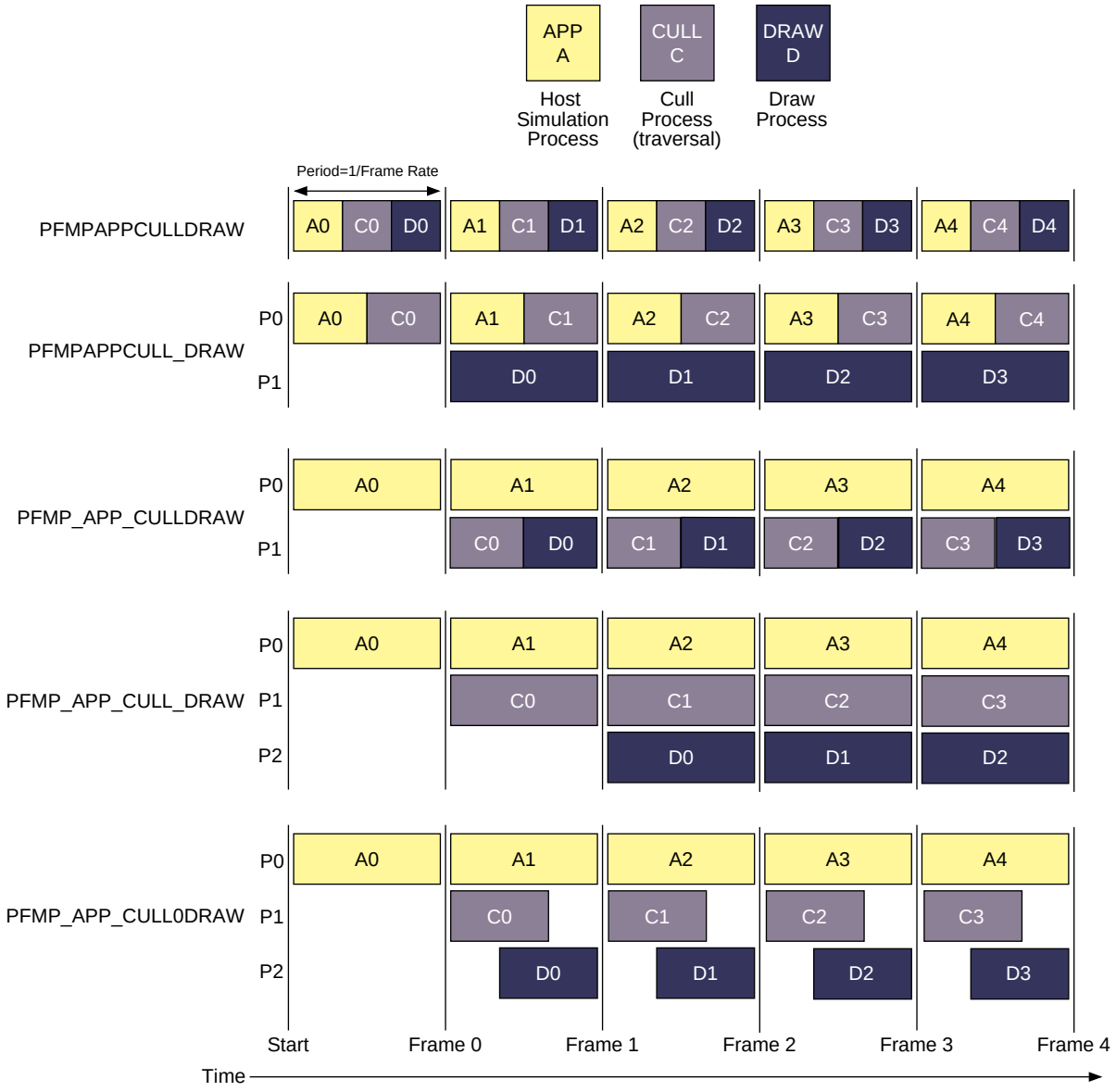


Figure 7-5 Multiprocessing Models

Notice that when a stage is split into its own process, the amount of time available for all stages increases. For example, in the case where the application, cull, and draw stages are 3 separate processes, it is possible for total system performance to be tripled over the single process configuration.

Asynchronous Database Processing

Many databases are too large to fit into main memory. A common solution to this problem is called *database paging* where the database is divided into manageable chunks on disk and loaded into main memory when needed. Usually chunks are paged in just before they come into view and are deleted from the scene when they are comfortably out of viewing range.

All this paging from disk and deleting from main memory takes a lot of time and is certainly not amenable to maintaining a fixed frame rate. The solution supported by IRIS Performer is *asynchronous database paging* in which a process, completely separate from the main processing pipeline(s), handles all disk I/O and memory allocations and deletions. To facilitate asynchronous database paging, IRIS Performer provides the pfBuffer structure and the DBASE process.

DBASE Process

The database (or DBASE) process is forked by **pfConfig()** if the PFMP_FORK_DBASE bit was set in the mode given to **pfMultiprocess()**. The database process is triggered when the application process calls **pfFrame()** and invokes the user-defined callback set with **pfDBaseFunc()**. The database process is totally asynchronous. If it exceeds a frame time it does not slow down any rendering or intersection pipelines.

The DBASE process is intended for asynchronous database management when used with pfBuffer.

pfBuffer

A pfBuffer is a logical buffer which isolates database changes to a single process, avoiding disastrous collisions on data from multiple processes. In typical use, a pfBuffer is created with **pfNewBuffer()**, made current with **pfSelectBuffer()** and merged with the main IRIS Performer buffer with

pfMergeBuffer(). While the DBASE process is intended for pfBuffer use, other processes forked by the application may also use different pfBuffers in parallel for multithreaded database management. By ensuring that only a single process uses a given pfBuffer at a given time and following a few scoping rules discussed below, the application can safely and efficiently implement asynchronous database paging

A pfNode is said to have *buffer scope* or be "in" a particular pfBuffer. This is an important concept because it affects what you can do with a given node. A newly-created node is automatically "in" the currently active pfBuffer until that pfBuffer is merged using **pfMergeBuffer()**. At that instant, the pfNode is moved into the main IRIS Performer buffer, otherwise known as the *application buffer*.

A rule in pfBuffer management is that a process may only access nodes that are in its current pfBuffer. As a result, a database process may not directly add a newly created subgraph of nodes to the main scene graph because all nodes in the main scene graph have application buffer scope only—they are isolated from the database pfBuffer. This may seem inconvenient at first but it eliminates catastrophic errors like, for example, the application process traverses a group at the same time you add a child, changing its child list and causing the traversal to chase a bad pointer.

Remedies to the inconveniences stated above are the **pfBufferAddChild()**, **pfBufferRemoveChild()** and **pfBufferClone()** routines. The first two routines are identical to their non-buffer counterparts **pfAddChild()** and **pfRemoveChild()** except the buffer versions do not happen immediately. Other functions, **pfBufferAdd()**, **pfBufferInsert()**, **pfBufferReplace()**, and **pfBufferRemove()**, perform the buffer-oriented delayed-action versions of the corresponding non-buffer pfList functions. In all cases the add, insert, replace, or removal request is placed on a list in the current pfBuffer and is processed later at **pfMergeBuffer()** time.

pfBufferClone() supports the notion of maintaining a "library" of common objects like trees or houses in a special library pfBuffer. The main database process then clones objects from the library pfBuffer into the database pfBuffer, possibly **pfFlatten()i**ng them for improved rendering performance. **pfBufferClone()** is identical to **pfClone()** except the buffer version requires that the source pfBuffer be specified and that all cloned nodes have scope in the source pfBuffer.

pfAsyncDelete

We've discussed how to create subgraphs for database paging: create and select a current pfBuffer, create nodes and build the subgraph, call pfBufferAddChild() and finally pfMergeBuffer() to incorporate the subgraph into the application's scene. But what about freeing the memory of old, unwanted subgraphs? For this we turn to **pfAsyncDelete()**.

pfDelete() is the normal mechanism for deleting objects and freeing their associated memory. However, **pfDelete()** can be a very costly routine since it must traverse, unreference, and register a deletion request for every IRIS Performer object it encounters which has a 0 reference count. **pfAsyncDelete()**, in conjunction with a forked DBASE process, moves the burden of deletion to the asynchronous database process so that all rendering and intersection pipelines are not adversely affected.

pfAsyncDelete() may be called from any process and places an asynchronous deletion request on a global list that is processed later by the DBASE stage when its trigger routine, **pfDBase()** is called. A major difference from **pfDelete()** is that **pfAsyncDelete()** does not immediately check the reference count of the object to be deleted and so does not return a value indicating whether the deletion was successful or not. At this time there is no way of querying the result of a **pfAsyncDelete()** request so care should be taken that the object to be deleted has no reference counts or memory leaks will result.

Rules for Invoking Functions While Multiprocessing

There are some restrictions on which functions can be called from an IRIS Performer process while multiple processes are running. Some specialized processes (such as the process handling the draw stage) can call only a few specific IRIS Performer functions, and can't call any other kinds of functions. This section lists general and specific rules concerning function invocation in the various IRIS Performer and user processes.

In this section, the term "the draw process" refers to whichever process is handling the draw stage, regardless of whether that process is also handling other stages. Similarly, "the cull process" and "the application process" refer to the processes handling the cull and application stages, respectively.

This is a general list of the kinds of routines you can call from each process:

application	configuration routines, creation and deletion routines, set and get routines, and trigger routines such as **pfAppFrame()**, **pfSync()**, **pfFrame()**
database	creation and deletion routines, set and get routines, **pfDBase()**, **pfMergeBuffer()**
cull	**pfCull()**, **pfCullPath()**, IRIS Performer graphics routines
draw	**pfClearChan()**, **pfDraw()**, **pfDrawChanStats()**, IRIS Performer graphics routines, graphics library routines

More specific elaborations:

- You should call configuration routines only from the application process, and only after **pfInit()** and before **pfConfig()**. **pfInit()** must be the first IRIS Performer call except for those routines that configure shared memory (see "Memory Allocation" in Chapter 10). Configuration routines don't take effect until **pfConfig()** is called. These are the configuration routines:
  - **pfMultipipe()**
  - **pfMultiprocess()**
  - **pfMultithread()**
  - **pfHyperpipe()**
- You should call creation routines, such as **pfNewChan()**, **pfNewScene()**, and **pfAllocIsectData()**, only in the application process after calling **pfConfig()** or in a process which has an active pfBuffer. There is no restriction on creating *libpr* objects like pfGeoSets and pfTextures.
- **pfDelete()** should only be called from the application or database processes. **pfAsyncDelete()** may be called from any process.
- Read-only routines—that is, the **pfGet*()** functions—can be called from any IRIS Performer process. However, if a forked draw process queries a pfNode, the data returned will not be frame-accurate. (See "Multiprocessing and Memory" on page 226.)

- Write routines—functions that set parameters—should be called only from the application process or a process with an active pfBuffer. It is possible to call a write routine from the cull process, but it isn't recommended since any modifications to the database will not be visible to the application process if it is separate from the cull (as when using PFMP_APP_CULLDRAW or PFMP_APP_CULL_DRAW). However, for transient modifications like custom level-of-detail switching, it is reasonable for the cull process to modify the database. The draw process should never modify any pfNode.
- IRIS Performer graphics routines should be called only from the cull or draw processes. These routines may modify hardware graphics state. They are the routines which can be captured by an open pfDispList. (See "Display Lists" in Chapter 10.) If invoked in the cull process, these routines are captured by an internal pfDispList and later invoked in the draw process; but if they are invoked in the draw process they immediately affect the current window. These graphics routines can be roughly partitioned into those that
  - apply a graphics entity: **pfApplyMtl()**, **pfApplyTex()**, and **pfLightOn()**
  - enable or disable a graphics mode: **pfEnable()**, **pfDisable()**
  - set or modify graphics state: **pfTransparency()**, **pfPushState()**, **pfMultMatrix()**
  - draw geometry or modify the screen: **pfDrawGSet()**, **pfDrawString()**, **pfClear()**
- Graphics library routines should be called only from the draw process. Since there is no open display list to capture these commands, an open window is required to accept them.

- "Trigger" routines should be called only from the appropriate processes (see Table 7-4).

Table 7-4 Trigger Routines and Associated Processes

Trigger Routine	Process/Context
pfAppFrame pfSync pfFrame	APP / main loop
pfPassChanData pfPassIsectData	APP / main loop
pfApp	APP / channel APP callback
pfCull pfCullPath	CULL / channel CULL callback
pfDraw pfDrawBin	DRAW / channel DRAW callback
pfNodeIsectSegs pfChanNodeIsectSegs	ISECT / callback or APP / main loop
pfDBase	DBASE / callback

- User-spawned processes created with sproc() can trigger parallel intersection traversals through multiple calls to **pfNodeIsectSegs()** and **pfChanNodeIsectSegs()**.
- Functions **pfApp()**, **pfCull()**, **pfDraw()**, and **pfDBase()** are only called from within the corresponding callback specified by **pfChanTravFunc()** or **pfDBaseFunc()**.

Multiprocessing and Memory

In IRIS Performer, as is often true of multiprocessing systems, memory management is the most difficult aspect of multiprocessing. Most data management problems in an IRIS Performer application can be partitioned into three categories:

- Memory visibility. IRIS Performer uses **fork()**, which—unlike **sproc()**—generates processes that don't share the same address space. The processes also cannot share global variables that are modified after the **fork()** call. After calling **fork()**, processes must communicate through explicit shared memory.
- Memory exclusion. If multiple processes read or write the same chunk of data at the same time, consequences can be dire. For example, one process might read the data while in an inconsistent state and end up dumping core while dereferencing a NULL pointer.
- Memory synchronization. IRIS Performer is configured as a pipeline where different processes are working on different frames at the same time. This pipelined nature is illustrated in Figure 7-5, which shows that, for instance, in the PFMP_APP_CULL_DRAW configuration the application process is working on frame *n* while the draw process is working on frame *n*-2. If, in this case, if we have only a single memory location representing the viewpoint, then it is possible for the application to set the viewpoint to that of frame *n* and the draw process to incorrectly use that same viewpoint for frame *n*-2. Properly synchronized data is called *frame accurate*.

Fortunately, IRIS Performer transparently solves all of the above problems for most IRIS Performer data structures and also provides powerful tools and mechanisms that the application can use to manage its own memory.

Shared Memory and pfInit()

As described in "Initializing and Configuring IRIS Performer" in Chapter 3, **pfInit()** creates a shared memory arena that is shared by all processes spawned by IRIS Performer and all user processes that are spawned from any IRIS Performer process. A handle to this arena is returned by **pfGetSharedArena()** and should be used as the arena argument to routines that create data that must be visible to all processes. Routines that accept an

arena argument are the **pfNew*()** routines found in the *libpr* library and the IRIS Performer memory allocator, **pfMalloc()**. In practice, it is usually safest to create *libpr* objects like pfGeoSets and pfMaterials in shared memory. *libpf* objects like pfNodes are always created in shared memory.

Allocating shared memory does not by itself solve the memory visibility problem discussed above. You must also make sure that the pointer that references the memory is visible to all processes. IRIS Performer objects, once incorporated into the database via routines like **pfAddGSet()**, **pfAddChild()**, and **pfChanScene()**, automatically ensure that the object pointers are visible to all IRIS Performer processes.

However, pointers to application data must be explicitly shared. A common way of doing this is to allocate the shared memory after **pfInit()** but before **pfConfig()** and to reference the memory with a global pointer. Since the pointer is set before **pfConfig()** forks any processes, these processes will all share the pointer's value and can thereby access the same shared memory region. However, if this pointer value changes in a process, its value will not change in any other process, since forked processes don't share the same address space.

Even with data visible to all processes, data exclusion is still a problem. The usual solution is to use hardware spin locks so that a process can lock the data segment while reading or writing data. If all processes must acquire the lock before accessing the data, then a process is guaranteed that no other processes will be accessing the data at the same time. All processes must adhere to this locking protocol, however, or exclusion isn't guaranteed.

In addition to a shared memory arena, **pfInit()** creates a semaphore arena whose handle is returned by **pfGetSemaArena()**. Locks can be allocated from this semaphore arena by **usnewlock()** and can be set and unset by **ussetlock()** and **usunsetlock()**, respectively.

pfDataPools

pfDataPools—named shared memory arenas with named allocation blocks—provide a complete solution to the memory visibility and memory exclusion problems, thereby obviating the need to set global pointers between **pfInit()** and **pfConfig()**. For more information about pfDataPools, see "Datapools" on page 388.

Passthrough Data

The techniques discussed thus far don't solve the memory synchronization problem. Performer's *libpf* library provides a solution in the form of *passthrough data*. When using pipelined multiprocessing, data must be passed through the processing pipeline so that data modifications reach the appropriate pipeline stage at the appropriate time.

Passthrough data is implemented by allocating a data buffer for each stage in the processing pipeline. Then, at well-defined points in time, the passthrough data is copied from its buffer into the next buffer along the pipeline. This copying guarantees memory exclusion, but you should minimize the amount of passthrough data to reduce the time spent copying.

Allocate a passthrough data buffer for the rendering pipeline using **pfAllocChanData()**; for data to be passed down the intersection pipeline, call **pfAllocIsectData()**. Data returned from **pfAllocChanData()** is passed to the channel cull and draw callbacks that are set by **pfChanTravFunc()**. Data returned from **pfAllocIsectData()** is passed to the intersection callback specified by **pfIsectFunc()**.

Passthrough data isn't automatically passed through the processing pipeline. You must first call **pfPassChanData()** or **pfPassIsectData()** to indicate that the data should be copied downstream. This requirement allows you to copy only when necessary—if your data hasn't changed in a given frame, simply don't call a **pfPass*()** routine, and you'll avoid the copy overhead. When you do call a **pfPass*()** routine, the data isn't immediately copied but is delayed until the next call to **pfFrame()**. The data is then copied into internal IRIS Performer memory and you're free to modify your passthrough data segment for the next frame.

Modifications to all *libpf* objects—such as pfNodes and pfChannels—are automatically passed through the processing pipeline, so frame-accurate behavior is guaranteed for these objects. However, in order to save substantial amounts of memory, *libpr* objects such as pfGeoSets and pfGeoStates don't have frame-accurate behavior; modifications to such objects are immediately visible to all processes. If you want frame-accurate modifications to *libpr* objects you must use the passthrough data mechanism, use a frame-accurate pfSwitch to select among multiple copies of the objects you want to change or use the pfCycleBuffer memory type described in "CycleBuffers" on page 389.

Chapter 8

“Creating Visual Effects”

This chapter describes special visual features such as lighting, backdrops, and fog.

Chapter 8

Creating Visual Effects

This chapter describes how to use environmental, atmospheric, lighting, and other visual effects to enhance the realism of your application.

Using pfEarthSky

A pfEarthSky is a special set of functions that clears a pfChannel's viewport efficiently and implements various atmospheric effects. A pfEarthSky is attached to a pfChannel with **pfChanESky()**. Several pfEarthSky definitions can be created, but only one can be in effect for any given channel at a time.

A pfEarthSky can be used to draw a sky and horizon, to draw sky, horizon, and ground, or just to clear the entire screen to a specific color and depth. The colors of the sky, horizon, and ground can be changed in real time to simulate a specific time of day. At the horizon boundary, the ground and sky share a common color, so that there is a smooth transition from sky to horizon color. The width of the horizon band can be defined in degrees.

A pfChannel's earth-sky model is automatically drawn by IRIS Performer before the scene is drawn unless the pfChannel has a draw callback set with **pfChanTravFunc()**. In this case it is the application's responsibility to clear the viewport. Within the callback **pfClearChan()** draws the channel's pfEarthSky.

Example 8-1 shows how to set up an **pfEarthSky()**.

Example 8-1 How to Configure a pfEarthSky

```
pfEarthSky  *esky;
pfChannel *chan;

sky = pfNewESky();
pfESkyMode(esky, PFES_BUFFER_CLEAR, PFES_SKY_GRND);
pfESkyAttr(esky, PFES_GRND_HT, -1.0f);
```

```
pfESkyColor(esky, PFES_GRND_FAR, 0.3f, 0.1f, 0.0f, 1.0f);
pfESkyColor(esky, PFES_GRND_NEAR, 0.5f, 0.3f, 0.1f,1.0f);
pfChanESky(chan, esky);
```

Atmospheric Effects

The complexities of atmospheric effects on visibility are approximated within IRIS Performer using a multiple-layer sky model, set up as part of the pfEarthSky function. In this design, individual layers are used to represent the effects of ground fog, clear sky, and clouds. Figure 8-1 shows the identity and arrangement of these layers.

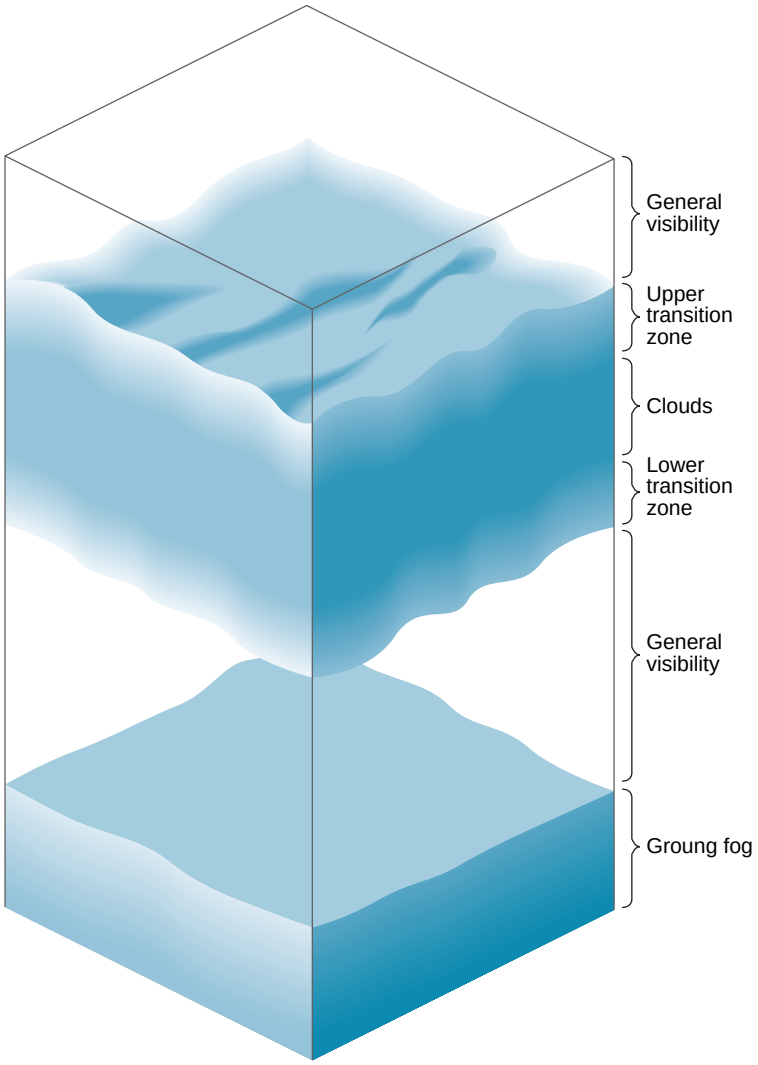


Figure 8-1 Layered Atmosphere Model

The lowest layer consists of ground fog, extending from the ground up to a user-selected altitude. The fog thins out with increasing altitude, disappearing entirely at the bottom of the general visibility layer. This layer extends from the top of the ground fog layer to the bottom of the cloud

layer's lower transition zone, if such a zone exists. The transition zone provides a smooth transition between general visibility and the cloud layer. (If there is no cloud layer, then general visibility extends upward forever.) The cloud layer is defined as an opaque region of near-zero visibility; you can set its upper and lower boundaries. You can also place another transition zone above the cloud layer to make the clouds gradually thin out into clear air.

Set up the atmospheric simulation with the commands listed in Table 8-1

Table 8-1 pfEarthSky Routines

Function	Action
pfNewESky	Create a pfEarthSky
pfESkyMode	Set the render mode
pfESkyAttr	Set the attributes of the earth and sky models
pfESkyColor	Set the colors for earth and sky and clear
pfESkyFog	Set the fog functions

You can set any pfEarthSky attribute, mode, or color in real time. Selecting the active pfFog definition can also be done in real time. However, changing the parameters of a pfFog once they are set isn't advised when in multiprocessing mode.

The default characteristics of a pfEarthSky are listed in Table 8-2.

Table 8-2 pfEarthSky Attributes

Attribute	Default
Clear method	PFES_FAST (full screen clear)
Clear color	0.0 0.0 0.0
Sky top color	0.0 0.0 0.44
Sky bottom color	0.0 0.4 0.7
Ground near color	0.5 0.3 0.0

Table 8-2 (continued) pfEarthSky Attributes

Attribute	Default
Ground far color	0.4 0.2 0.0
Horizon color	0.8 0.8 1.0
Ground fog	NULL (no fog)
General visibility	NULL (no fog)
Cloud top	20000.0
Cloud bottom	20000.0
Cloud bottom color	0.8 0.8 0.8
Cloud top color	0.8 0.8 0.8
Transition zone bottom	15000.0
Transition zone top	25000.0
Ground height	0
Horizon angle	10 degrees

By default, an earth-sky model isn't drawn. Instead, the channel is simply cleared to black and the Z-buffer is set to its maximum value. This default action also disables all other atmospheric attributes. To enable atmospheric effects, select PFES_SKY, PFES_SKY_GRND, or PFES_SKY_CLEAR when turning on the earth-sky model.

Clouds are disabled when the cloud top is less than or equal to the cloud bottom. Cloud transition zones are disabled when clouds are disabled.

Fog is enabled when either the general or ground fog is set to a valid pfFog. If ground fog isn't enabled, no ground fog layer will be present and fog will be used to support general visibility. Setting a fog attribute to NULL disables it. See "Fog" on page 350 for further information on fog parameters and operation.

The earth-sky model is an attribute of the channel and thus accesses information about the viewer's position, current field of view, and other pertinent information directly from pfChannel. To set the pfEarthSky in a channel, use **pfChanESky()**.

Light Points

pfLightPoint

Note: pfLightPoint nodes have been obsoleted in favor of the *libpr* attribute, pfLPointState described in "pfLPointState" on page 237. However, the pfLightPoint node is still available for convenience.

A pfLightPoint is a pfNode that contains one or more light points. Light points are used in flight simulation applications to simulate runway lighting, taxiway lights, and street lights. The light points in a pfLightPoint node share all attributes except point location. A pfLightPoint node can be thought of as representing a string of similar point lights.

pfNewLPoint() creates and returns a handle to a pfLightPoint node. **pfLPointSize()** sets the screen size for all points in a pfLightPoint. The size is specified in pixels and is used as the argument to the IRIS GL function **pntsizef()** or the OpenGL function **glPointSize()**.

Whenever possible, graphics library antialiased points are used to represent light points, but the actual representation depends on the graphics hardware being used. See the IRIS GL **pntsizef()** reference page or the OpenGL **glPointSize()** reference page for a description of available light point sizes on your IRIS hardware.

pfLPointColor() sets the color for a specified light point in the given pfLightPoint node. Each light in a pfLightPoint can be turned off by setting its red, green, and blue values to 0.0, or by setting its alpha value to 0.0 (which makes it completely transparent). Light points with a color of zero will not be considered for rendering.

pfLPointRot() and **pfLPointShape()** control the direction and shape of all the lights points in a pfLightPoint. The direction of the light points in *lpoint* is the positive Y axis, rotated about the X axis by *elev* and then rotated about the Z axis by *azim*. Roll affects only the light envelope. **pfLPointShape()** is used to set the intensity distribution of the light points (their visibility envelope). A pfLightPoint can have any one of three envelope shapes: omnidirectional, unidirectional, or bidirectional. If a light point is omnidirectional, it can be seen from any direction with the same intensity. This is the fastest light to process and draw using IRIS Performer.

Unidirectional and bidirectional lights have an elliptical cone of intensity. The intensity falloff from the center of the cone to the edge is exponential and is also set with **pfLPointShape()**.

pfLPointPos() is used to set the position of each light point in a pfLightPoint. A runway strobe could be implemented with a pfLightPoint containing a single point and repositioning that point at regular intervals. A rotating beacon could be created using a single point and rotating it each frame using **pfLPointRot()**.

pfLPointState

A pfLPointState is a libpr attribute which, when attached to an appropriately configured pfGeoState will cause pfGeoSets of type PFGS_POINTS to be rendered as light points. pfLPointStates are intended to obsolete pfLightPoints since they provide a more powerful mechanism for simulating light points. Special features of the pfLPointState include:

- Perspective point size -- points farther away look smaller than those closer to the eye.
- Perspective fading -- points whose computed size is less than an application-defined threshold are made more transparent, rather than shrinking the point any further.
- Fog punch-through.
- Directionality including bi-directional points with different front and back colors.

Since pfLPointState uses the pfGeoState mechanism, light points can appear different to different pfChannels by using indexed pfGeoStates and pfGeoState tables. For example, a light point may look brighter in an infra-red channel than in an out-the-window channel.

One improvement over pfLightPoints is that point directions are supplied by a pfGeoSet's normals so a single pfLPointState can be used for many light points with differing directions. A further improvement is in performance. pfLPointStates can use texture mapping to simulate either directionality or both perspective fading and fog punch-through. This frees the CPU from making these expensive calculations. Example 8-2 is a portion of the sample program, *lpstate.c* found in */usr/share/Performer/src/pguide/libpf/C*.

Example 8-2 How to set up a pfLPointState

```
#define NPOINTS2 30
#define NPOINTS (NPOINTS2 * (NPOINTS2 - 2))

static pfGeode*
initLPoints(void)
{
 pfLPointState    *lps;
 pfTexGen *tgen;
 pfTexture *tex;
 pfGeoState *gst;
 pfGeode *gd;
 pfGeoSet *gs;
 pfVec3     *norms, *colors, *coords;
 pfMatrix        squash, squashInvTransp;
 float     phi, dphi, theta, dtheta;
 int i, j, k;
 void        *arena = pfGetSharedArena();

 /*---- Set up pfLPointState ----*/

 lps = pfNewLPState(arena);

 /* Enable perspective size computation */
 pfLPStateMode(lps, PFLPS_SIZE_MODE, PF_ON);

 /* Clamp point size between .25 and 4 pixels */
 pfLPStateVal(lps, PFLPS_SIZE_MIN_PIXEL, .25f);
 pfLPStateVal(lps, PFLPS_SIZE_MAX_PIXEL, 4.0f);
```

```
 /* Real-world point size is .15 meters */
 pfLPStateVal(lps, PFLPS_SIZE_ACTUAL, .15f);

 /* Fade points smaller than 2 pixels */
 pfLPStateVal(lps, PFLPS_TRANSP_PIXEL_SIZE, 2.0f);

 /* Linear fade, scaled by .6 and alpha clamped at .1 */
 pfLPStateVal(lps, PFLPS_TRANSP_EXPONENT, 1.0f);
 pfLPStateVal(lps, PFLPS_TRANSP_SCALE, .6f);
 pfLPStateVal(lps, PFLPS_TRANSP_CLAMP, .1f);

 /* Points are fogged as if 4 times closer than they really
are */
 pfLPStateVal(lps, PFLPS_FOG_SCALE, .25f);

 /* Compute true, slant range from eye to points */
 pfLPStateMode(lps, PFLPS_RANGE_MODE, PFLPS_RANGE_MODE_TRUE);

 /* Points are bidirectional with purple back color */
 pfLPStateMode(lps, PFLPS_SHAPE_MODE,
                    PFLPS_SHAPE_MODE_BI_COLOR);
 pfLPStateBackColor(lps, 1.f, 0.0f, 1.f, 1.0f);

 /*
 * Point shape is 15 horiz and 60 degrees vertical with
 * no roll, falloff of 1 and ambient intensity of .1
 */
 pfLPStateShape(lps, 15.0f, 60.0f, 0.0f, 1, .1f);

 /*---- Set up pfGeoState ----*/

 gst = pfNewGState(arena);

 /* Specify high-quality transparency */
 pfGStateMode(gst, PFSTATE_TRANSPARENCY,
         PFTR_BLEND_ALPHA | PFTR_NO_OCCLUDE);
 pfGStateVal(gst, PFSTATE_ALPHAREF, 0.0f);
 pfGStateMode(gst, PFSTATE_ALPHAFUNC, PFAF_GREATER);

 pfGStateMode(gst, PFSTATE_ENFOG, 0);
 pfGStateMode(gst, PFSTATE_ENLIGHTING, 0);
 pfGStateMode(gst, PFSTATE_ENTEXTURE, 1);
 pfGStateMode(gst, PFSTATE_ENLPOINTSTATE, 1);
 pfGStateAttr(gst, PFSTATE_LPOINTSTATE, lps);
```

```
 /*---- Configure texturing ----*/

 tgen = pfNewTGen(arena);
 tex = pfNewTex(arena);

#define USE_TEXTURE_MAPPING_FOR_DIRECTIONALITY
#ifdef USE_TEXTURE_MAPPING_FOR_DIRECTIONALITY
 /*
 * Use texture mapping for directionality. CPU computes size
 * and range and fog attenuation
 */
 pfLPStateMode(lps, PFLPS_DIR_MODE, PFLPS_DIR_MODE_TEX);
 pfLPStateMode(lps, PFLPS_TRANSP_MODE,
                    PFLPS_TRANSP_MODE_ALPHA);
 pfLPStateMode(lps, PFLPS_FOG_MODE, PFLPS_FOG_MODE_ALPHA);
 pfuMakeLPStateShapeTex(lps, tex, 256);
 pfGStateAttr(gst, PFSTATE_TEXTURE, tex);
 pfTGenMode(tgen, PF_S, PFTG_SPHERE_MAP);
 pfTGenMode(tgen, PF_T, PFTG_SPHERE_MAP);
#else
 /*
 * Use texture mapping for range and fog attenuation. CPU
 * computes size and directionality.
 */
 pfLPStateMode(lps, PFLPS_DIR_MODE, PFLPS_DIR_MODE_ALPHA);
 pfLPStateMode(lps, PFLPS_TRANSP_MODE,
                    PFLPS_TRANSP_MODE_TEX);
 pfLPStateMode(lps, PFLPS_FOG_MODE, PFLPS_FOG_MODE_TEX);
 pfuMakeLPStateRangeTex(lps, tex, 256, pfNewFog(NULL));
 pfGStateAttr(gst, PFSTATE_TEXTURE, tex);
 pfTGenPlane(tgen, PF_S, 0.0f, 0.0f, 1.0f, 0.0f);
 pfTGenPlane(tgen, PF_T, 0.0f, 0.0f, 1.0f, 0.0f);
 pfTGenMode(tgen, PF_S, PFTG_EYE_PLANE_IDENT);
 pfTGenMode(tgen, PF_T, PFTG_EYE_PLANE_IDENT);
#endif
 pfGStateAttr(gst, PFSTATE_TEXGEN, tgen);
 pfGStateMode(gst, PFSTATE_ENTEXGEN, 1);

 /* Make PFGS_POINTS pfGeoSet arranged in a sphere */

 gd = pfNewGeode();
 gs = pfNewGSet(arena);
 pfGSetPrimType(gs, PFGS_POINTS);
 pfGSetNumPrims(gs, NPOINTS);
```

```
colors = pfMalloc(sizeof(pfVec3) * NPOINTS, arena);
coords = pfMalloc(sizeof(pfVec3) * NPOINTS, arena);
norms = pfMalloc(sizeof(pfVec3) * NPOINTS, arena);
pfGSetAttr(gs, PFGS_NORMAL3, PFGS_PER_VERTEX, norms, NULL);
pfGSetAttr(gs, PFGS_COLOR4, PFGS_PER_VERTEX, colors, NULL);
pfGSetAttr(gs, PFGS_COORD3, PFGS_PER_VERTEX, coords, NULL);

pfGSetGState(gs, gst);
pfAddGSet(gd, gs);

/* Squash sphere into an ellipse so perspective point size
   is more easily seen
*/
pfMakeRotMat(squash, 90.0f, 1.0f, 0.0f, 0.0f);
pfPostScaleMat(squash, squash, 1.0f, 2.0f, .5f);
pfInvertAffMat(squashInvTransp, squash);
pfTransposeMat(squashInvTransp, squashInvTransp);

dphi = 180.0f / (NPOINTS2-1);
dtheta = 360.0f / NPOINTS2;

phi = dphi;
for (k=0, i=0; i<NPOINTS2 - 2; i++)
{
   float    ct, st, sp, cp;

   theta = 0.0f;
   pfSinCos(phi, &sp, &cp);

   for (j=0; j<NPOINTS2; j++, k++)
   {
        pfSetVec4(colors[k], 1.0f, 1.0f, 1.0f, 1.0f);

        pfSinCos(theta, &st, &ct);
        pfSetVec3(norms[k], ct * sp, st * sp, cp);
        pfScaleVec3(coords[k], 10.0f, norms[k]);

        pfXformPt3(coords[k], coords[k], squash);
        pfXformVec3(norms[k], norms[k], squashInvTransp);
        pfNormalizeVec3(norms[k]);

        theta += dtheta;
    }

    phi += dphi;
```

```
 }
 return gd;
}
```

Spotlights and Shadows

A pfLightSource node's primary purpose is to represent a graphics library light source (pfLight) in a scene graph. The position and orientation of the light source is affected by transformations inherited through the scene graph providing a simple means of moving lights about. In conjunction with an object's material properties (pfMaterial) and the global lighting model (pfLightModel), illumination is computed at geometry vertices by the Geometry Pipeline.

While this kind of "bread-and-butter" lighting is very efficient it does not adequately simulate certain important lighting effects. In particular, per-pixel spotlights and shadows are not supported by the default lighting mechanism. On graphics library implementations which support texture mapping, the pfLightSource node supports per-pixel spotlights and *opera lighting* through a technique called *projective texturing* and also support shadows if *shadow map* hardware is available. In practice, default lighting is used in conjunction with projected texturing since the latter does not consider geometry normals but the former does.

Both projected texturing and shadows require a pfFrustum which defines the "viewing volume" of the light source. The frustum's eye point is located at the pfLightSource position and the frustum's view direction is aligned along the pfLightSource's +Y axis. You can think of the frustum as a slide projector whose eyepoint is the projector's bulb and whose view plane is the slide holder and the projected texture map is the slide. A pfLightSource's frustum is set with **pfLSourceAttr()** with the PFLS_PROJ_FRUSTUM identifying token.

In addition to a frustum, projected texturing requires a texture map supplied with **pfLSourceAttr()** with PFLS_PROJ_TEX identifying token. This texture map can be anything but is usually a texture with identical intensity and alpha components which represent the soft-edged circle of a spotlight. Colored spotlights are simulated by coloring the pfLightSource with **pfLSourceColor()** rather than coloring the texture map. Opera lighting, where an actual color slide is projected onto the set, is simulated with a

texture map consisting of red, green, blue, and alpha components (this is supported only if the pfLightSource is the only one that is projecting a texture).

Shadows do not require the application to supply a texture map; rather, one is automatically created by IRIS Performer. However, the size of the shadow map has an important effect on the quality of the shadows and may be set with the PFLS_SHADOW_SIZE token to **pfLSourceVal()**. For high quality shadows, the pfLightSource's frustum must encompass the entire scene that is to be shadowed as tightly as possible. If the frustum's field-of-view or far to near clipping plane ratio becomes too large, the shadows will become blocky and incorrect. One way to improve shadow quality is to increase the size of the shadow map but this will decrease performance.

Projected textures and shadows both use texture mapping so if your scene is texture mapped, multiple rendering passes are required to combine the projected texture with the normal scene textures. In **pfDraw()**, pfChannels assume the scene is textured and automatically render the scene multiple times. However, if your scene is not textured you can avoid a complete rendering pass by setting the PFMPASS_NONTEX_SCENE bit in the PFTRAV_MULTIPASS mode given to **pfChanTravMode()**.

Example 8-3 is a code snippet which shows how to set up both projected texture and shadowing pfLightSources.

Example 8-3 Projected texture and shadow pfLightSources

```
pfFrustum *shadFrust;
pfLightSource *shad, *proj;
pfDCS *shadDCS, *projDCS;
pfTexture *tex;

// Create and configure shadow frustum
shadFrust = pfNewFrust(arena);
pfMakeSimpleFrust(shadFrust, 60.0f);
pfFrustNearFar(shadFrust, 1.0f, 100.0f);

/* Create and configure red shadow casting light source */
shad = pfNewLSource();
pfLSourceMode(shad, PFLS_SHADOW_ENABLE, 1);
pfLSourceAttr(shad, PFLS_PROJ_FRUST, shadFrust);
pfLSourceColor(shad, PFLT_DIFFUSE, 1.0f, 0.0f, 0.0f);
pfLSourceVal(shad, PFLS_INTENSITY, .5f);
```

```
pfLSourcePos(shad, 0.0f, 0.0f, 0.0f, 1.0f); /* make local */

/* Create and configure blue spotlight */
proj = pfNewLSource();
tex = pfNewTex(arena);
pfLoadTexFile(tex, “spotlight.inta”);
pfLSourceMode(proj, PFLS_PROJTEX_ENABLE, 1);
pfLSourceAttr(proj, PFLS_PROJ_FRUST, shadFrust);
pfLSourceAttr(proj, PFLS_PROJ_TEX, tex);
pfLSourceColor(proj, PFLT_DIFFUSE, 0.0f, 0.0f, 1.0f);
pfLSourceVal(proj, PFLS_INTENSITY, .5f);
pfLSourcePos(proj, 0.0f, 0.0f, 0.0f, 1.0f); /* make local */

/* Add to DCSes so we can move lights around */
shadDCS = pfNewDCS();
pfAddChild(shadDCS, shad);
pfAddChild(scene, shadDCS);

projDCS = pfNewDCS();
pfAddChild(projDCS, proj);
pfAddChild(scene, projDCS);
```

For scenes with multiple pfLightSources you can scale the contribution from each source with the PFLS_INTENSITY value so that the total illumination from all sources does not exceed one. Otherwise lighted geometry may become saturated and not look real.

pfLightSources which are located near the eyepoint and which project a texture to simulate spotlights can attenuate the spotlight based on range by using a pfFog to define the falloff ramp. The pfFog is set with the PFLS_PROJ_FOG token to **pfLSourceAttr()** and should have the same color as the lighting ambient.

Morphing

Morphing is the smooth transition from one particular appearance to another. Morphing can refer to images or geometry colors, texture coordinate or coordinates. For example, you could “morph” the following:

- the image of a person’s face into that of another person
- color to simulate a flickering fire

- texture coordinates to simulate rippling ocean waves
- coordinates to make a 3D model of a face smile or frown.

pfMorph is a group node that supports the morphing of lists of numbers. These numbers could be colors, coordinates, or pixels -- it is up to you to decide what to morph.

Conceptually, pfMorph combines multiple input lists of numbers into a single output list. This output may be used arbitrarily but is most often used as a pfGeoSet attribute array, be it colors, normals, texture coordinates, or coordinates. The combination is linear, that is, the input lists are scaled by a value (which has no restricted range) and then summed together to produce the output. One example of geometric morphing is in facial animation where multiple input faces with canonical expressions: smiley, frowny, surprised, are combined to produce a face with a mixture of expressions.

pfMorphs can handle multiple sets of input->output lists so a single pfMorph could, for example, morph the normals and coordinates of an animated figure. An input->output set is called a *morph attribute* since a set typically corresponds to a pfGeoSet attribute. Morph attributes are specified with **pfMorphAttr()**. Example 8-4 is a snippet of the morph.c test program in */usr/share/Performer/src/pguide/libpf/C/* shows how to configure a pfMorph node that morphs the normals and coordinates of a pfGeoSet.

Example 8-4 How to set up a pfMorph node.

```
/* Sinusoidally modify morph weights to oscillate between
* cube and sphere.
*/
static void
breatheMorph(pfMorph *morph, double t)
{
 float     s = (sinf(t) + 1.0f) / 2.0f;
 float     weights[2];

 weights[0] = s;
 weights[1] = 1.0f - s;

 pfMorphWeights(morph, 0, weights); /* coordinate weights*/
 pfMorphWeights(morph, 1, weights); /* normal weights */
}
```

```
static pfMorph*
initMorph(void)
{
 pfGeoSet     *gset;
 pfGeode     *geode;
 pfGeoState     *gstate;
 pfMaterial     *mtl;
 pfMorph     *morph;
 ushort     *icoords, *inorms;
 pfVec3     *coords, *ncoords, *norms, *nnorms;
 float     *srcs[2];
 int          i, nSph;
 void     *arena = pfGetSharedArena();

 morph = pfNewMorph();
 geode = pfNewGeode();
 gset = pfdNewSphere(400, arena);
 gstate = pfNewGState(arena);

 mtl = pfNewMtl(arena);
 pfMtlColor(mtl, PFMTL_DIFFUSE, 1.0f, 0.0f, 0.0f);
 pfMtlColor(mtl, PFMTL_SPECULAR, 1.0f, 1.0f, 1.0f);
 pfMtlColorMode(mtl, PFMTL_BOTH, PFMTL_CMODE_OFF);
 pfMtlShininess(mtl, 32);

 pfGStateAttr(gstate, PFSTATE_FRONTMTL, mtl);
 pfGStateMode(gstate, PFSTATE_ENLIGHTING, 1);
 pfGSetGState(gset, gstate);

 pfAddGSet(geode, gset);
 pfAddChild(morph, geode);

 /*
 * NULL forces recomputation of bound. Force it to be static
 * to avoid expensive recomputation. Static bound should
 * encompass the extent of all morph possibilities.
 */
 pfGSetBBox(gset, NULL, PFBOUND_STATIC);
 pfNodeBSphere(geode, NULL, PFBOUND_STATIC);

 pfGetGSetAttrLists(gset, PFGS_COORD3, (void**)&coords,
&icoords);
 pfGetGSetAttrLists(gset, PFGS_NORMAL3, (void**)&norms,
&inorms);
 nSph = pfGetSize(coords) / sizeof(pfVec3);
```

```
ncoords = pfMalloc(pfGetSize(coords), arena);
nnorms = pfMalloc(pfGetSize(norms), arena);

for (i=0; i<nSph; i++)
{
    int     max;
    float     t;

    /* Find which face of the cube this vertex maps to */
    if (PF_ABS(coords[i][PF_X]) > PF_ABS(coords[i][PF_Y]))
    {
     if (PF_ABS(coords[i][PF_X]) > PF_ABS(coords[i][PF_Z]))
         max = PF_X;
     else
         max = PF_Z;
    }
    else
    {
     if (PF_ABS(coords[i][PF_Y]) > PF_ABS(coords[i][PF_Z]))
         max = PF_Y;
     else
         max = PF_Z;
    }

    /* Compute cube normals and coordinates */
    pfSetVec3(nnorms[i], 0.0f, 0.0f, 0.0f);
    if (coords[i][max] < 0.0f)
    {
     t = -1.0f / coords[i][max];
     pfScaleVec3(ncoords[i], t, coords[i]);
     nnorms[i][max] = -1.0f;
    }
    else
    {
     t = 1.0f / coords[i][max];
     pfScaleVec3(ncoords[i], t, coords[i]);
     nnorms[i][max] = 1.0f;
    }
}

/* Morph attribute 0 is coordinates */
srcs[0] = (float*)coords;
srcs[1] = (float*)ncoords;
pfMorphAttr(morph, 0, 3, nSph, NULL, 2, srcs, NULL, NULL);
```

```
 pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX,
      (void*)pfGetMorphDst(morph, 0), icoords);

 /* Morph attribute 1 is normals */
 srcs[0] = (float*)norms;
 srcs[1] = (float*)nnorms;
 pfMorphAttr(morph, 1, 3, nSph, NULL, 2, srcs, NULL, NULL);

 pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_VERTEX,
      (void*)pfGetMorphDst(morph, 1), inorms);

 return morph;
}
```

Chapter 9

“Importing Databases”

This chapter describes a variety of database formats and their corresponding conversion utilities.

Chapter 9

Importing Databases

Overview of Performer Database Creation and Conversion

Once you've learned how to create visual simulation applications with IRIS Performer your next task is to import visual databases into those applications. Import and export functions for more than 30 popular database formats are provided with IRIS Performer to ease this effort. This chapter describes the steps involved in creating custom loaders for other data formats and then reviews each of these pre-existing file-loading utilities. Along the way, it also describes several utility functions in the IRIS Performer database utility library that can make the process of database conversion easier for you.

Source code is provided for most of the tools discussed in this chapter. In most cases the loaders are short, easy to understand, and easy to modify. Table 9-1 lists the subdirectories of */usr/share/Performer/src/lib* where you can find the source code for the database processing tools.

Table 9-1 Database-Importer Source Directories

Directory Name	Directory Contents
libpfdu	General database processing tools and utilities
libpfdb	Load, convert, and store specific database formats
libpfutil	Additional utility functions

Before you can import a database, you must create it. Some simulation applications create data procedurally; for examples of this approach, see the "Silicon Graphics PHD Format" on page 295 or "Sierpinski Sponge Format" sections of this chapter. In most cases, however, you must create visual databases manually. Several software packages are available to help with this task, and most such systems facilitate geometric modeling, texture creation, and interactive specification of colors and material properties.

Some advanced systems support level-of-detail specification, animation sequences, motion planning for jointed objects, automated roadway and terrain generation, and other specialized functions.

libpfdu - Utilities for Creation of Efficient Performer Run-Time structures

There are several layers of support in IRIS Performer for loading 3-D models and 3-D environments into IRIS Performer run-time scene graphs. In fact, IRIS Performer 2.0 contains the new *libpfdu* library devoted to the import of data into (and export of data from) IRIS Performer run-time structures. Note that two database exporters have already been written for the Medit and DWB database formats.

At the top level of the API, IRIS Performer provides a standard set of functions to read in files and convert databases of unknown type. This functionality is centered around the notion of a database converter. A database converter is an abstract entity that knows how to perform some or all of a set of database format conversion functions with a particular database format. Moreover, converters must follow certain API guidelines for standard functionality such that they can be easily integrated into IRIS Performer in a run-time environment without IRIS Performer needing any *a priori* knowledge of a particular converter's existence. This run-time integration is done through the use of dynamic shared object (DSO) libraries.

pfdLoadFile - Loading Arbitrary Databases into Performer

libpfdu provides the following general routines to operate on 3-D databases.

Table 9-2 libpfdu database converter functions

Function Name	Description
pfdLoadFile	Load a database file into an IRIS Performer scene graph
pfdStoreFile	Store a run-time scene graph into a database file

Table 9-2 (continued) libpfdu database converter functions

Function Name	Description
pfdConvertFrom	Convert an external run-time format into an IRIS Performer scene graph
pfdConvertTo	Convert an IRIS Performer scene graph into an external run-time format

The database loader utility library, *libpfdu*, provides a convenient function named **pfdLoadFile()** that can import database files stored in any supported format - see Table 9-6 for a list of supported formats.

Loading database files with **pfdLoadFile()** is easy. The function prototype is

```
pfNode *pfdLoadFile(char *fileName);
```

pfdLoadFile() tests the filename-extension portion of *fileName* (the substring starting at the last period in *fileName*, if any) for one of the format-name codes listed in Table 9-6, then calls the appropriate importer.

The file-format selection process is implemented using dynamic loading of DSOs, which are IRIX Dynamic Shared Objects. This process allows new loaders that are developed as database formats change to be used with IRIS Performer-based applications without requiring recompilation of the IRIS Performer application. The details of the loading process internal to **pfdLoadFile()** include searching for the named file using the current IRIS Performer file path, extraction of the file-type extension, translation of the extension using a registered alias facility, formation of the DSO name, searching through a list of user-defined and standard directories for that DSO, dynamic loading of the indicated DSO using **dlopen()**, formation of a loader function name, finding that function within the DSO using **dlsym()**, and finally, invocation of the loader function.

The loader function name is constructed from two components, a prefix and the loader suffix. The prefix is always “pfdLoadFile_” and the suffix is simply the file extension string. Examples of several complete loader function names are shown in Table 9-3.

Table 9-3 Loader Name Composition

File Extension	Loader Function Name
dwb	pfdLoadFile_dwb
flt	pfdLoadFile_flt
medit	pfdLoadFile_medit
obj	pfdLoadFile_obj

Several shell environment variables are used in the loader location process. These are PFLD_LIBRARY_PATH, LD_LIBRARY_PATH, and PFHOME. Confusion about loader locations can be resolved by consulting the sources mentioned above to understand the use of these directory lists and reading the following section, “Database Loading Details” on page 254. When the pfNotifyLevel is set to PFNFY_DEBUG or greater, the DSO and loader function names are printed as database are loaded, as is the name of each directory that is searched for the DSO.

The IRIS Performer sample programs, including *perfly*, use **pfdLoadFile()** for database importing. This allows them to simultaneously load and display databases in many disparate formats. As you develop your own database loaders, follow the source code examples in any of the *libpfdb* loaders. Then you will be able to load your data into any IRIS Performer application. You will not need to rebuild *perfly* or other applications to view your databases.

Database Loading Details

Details about the database loading process are described further in this section, the pfdLoadFile reference page, and the source code which is in */usr/share/Performer/src/lib/libpfdu/pfdLoadFile.c*.

pfdLoadFile(), **pfdStoreFile()**, **pfdConvertFrom()**, and **pfdConvertTo()** exist only as a level of indirection to allow a user to manipulate all databases

regardless of format through a central API. They are in fact merely a mechanism for creating an open environment for data sharing among the multitudes of 3-dimensional database formats. In fact, each of these routines merely determines via file type extension which database converter to load as a run-time DSO and then calls the appropriate functionality from that converter's DSO. All converters must provide API that is exactly the same as the corresponding libpfdu API with _EXT added to the routine names (for example, for "*.medit*" files, the suffix is "_medit"). Note that multiple physical extensions can be mapped to one converter extension via calls to **pfdAddExtAlias()**. Several aliases are pre-defined upon initialization of *libpfdu*. For example, VRML ".wrl" files are mapped to Inventor ".iv" so that the Open Inventor 2.1 loader can be used to import them

It is also important to note that because each of these converters are unique entities that they each may have state that is important to their proper function. Moreover, their database format may allow for multiple IRIS Performer interpretations and so there exists API (see Table 9-4) not only to initialize and exit database converters, but also to set and get modes, attributes, and values that might affect the converter's methodology:

Table 9-4 libpfdu database converter management functions

Function Name	Description
pfdInitConverter	Initialize a database conversion DSO
pfdExitConverter	Exit a database conversion DSO
pfdConverterMode	Specify a mode for a specific conversion DSO
pfdGetConverterMode	Get a mode setting from a specific conversion DSO
pfdConverterAttr	Specify an attribute for a conversion DSO
pfdGetConverterAttr	Get an attribute setting from a conversion DSO
pfdConverterVal	Specify a value for a conversion DSO
pfdGetConverterVal	Get a value setting from a conversion DSO

Once again each converter provides the equivalent routines with _EXT added to the function name.

For example, the converter for the Open Inventor format would define the function **pfdInitConverter_iv()** if it needed to be initialized before it was used. Likewise, it would define the function **pfdLoadFile_iv()** to read an Open Inventor "*.iv*" file into an IRIS Performer scene graph.

Note: Because each converter is an individual entity (DSO) and deals with a particular type of database, it may be the case that a converter will *not* provide all of the functionality listed above, but rather only a subset. For instance, most converters that come with IRIS Performer 2.0 only implement their version of **pfdLoadFile** but not **pfdStoreFile**, **pfdConvertFrom**, or **pfdConvertTo**. However, users are free to add this functionality to the converters via compliant API and IRIS Performer's *libpfdu* will immediately recognize this functionality. Also, *libpfdu* traps access to non-existent converter functionality and returns gracefully to the calling code while notifying the user that the functionality could not be found.

Finding and initializing a Converter

When one of the general database converter functions is called, it in turn calls the corresponding routine provided by the converter, passing on the arguments it was given.

But the first time a converter is called, a search occurs to identify the converter and the functions it provides. This is accomplished as follows.

- Parse the extension - what appears after the final "." in the filename. This is referred to as EXT in the following bulleted items.
- Check to see if any alias was created for the EXT extension with **pfdAddExtAlias()**. If a translation is defined, EXT is replaced with that extension.
- Check the current executable to see if the symbol pfdLoadFile_EXT is already defined, i.e. if the loader was statically linked into the executable or a DSO was previously loaded by some other mechanism. If not, the search continues.
  - Generate a DSO library name to search for using on the extension prototype "libpfEXT_{igl,ogl}{-g,}.so". This means the following strings will be constructed based upon whether OpenGL or IRIS GL is being used with IRIS Performer:

    libpfEXT_igl.so for the optimized IRIS GL loader

libpfEXT_igl-g.so for the debug IRIS GL loader

libpfEXT_ogl.so for the optimized OpenGL loader

libpfEXT_ogl-g.so for the debug OpenGL loader

- Look for the DSO in several places including:

```
.
$PFLD_LIBRARY_PATH
$LD_LIBRARY_PATH
$PFHOME/usr/lib{,32,64}/libpfdb
$PFHOME/usr/share/Performer/lib/libpfdb
```

- Open the DSO via **dlopen()**.

- Once the object has been found, processing continues.
  - Query all libpfdu converter functionality from the symbol table of the DSO using **dlsym()** with function names generated by appending _EXT to the name of the corresponding pfd routine name. This symbol dictionary is retained for future use.
  - Invoke the converter's initialization function, **pfdInitConverter_EXT()**, if it exists.

Developing Custom Importers

Having fully described how database converters can be integrated into IRIS Performer and the types of functionality they provide, the next undertaking is actually implementing a converter from scratch. IRIS Performer 2.0 makes a great effort at allowing the quick and easy development of effective and efficient database converters.

While creating a new file loader for IRIS Performer isn't inherently difficult, it does require a solid understanding of the following issues:

- The structure and interpretation of the data file to be read
- The scene graph concepts and nodes of *libpf*
- The geometry and attribute definition objects of *libpr*

Structure and interpretation of the Database File Format

In order to effectively convert a database into an IRIS Performer scene graph it is important to have a substantial understanding of several concepts related to the original database format:

- the parsing of the file based on the database format
- the data types represented in the format and their IRIS Performer correspondence
- the scene graph structure of the file (if any)
- the method of graphics state definition and inheritance defined in the format.

Before trying to convert sophisticated 3-D database formats into IRIS Performer it is important to have a thorough grasp of how every structure in the format needs to affect how IRIS Performer performs its run-time management of a scene graph. However, although it requires a great deal of understanding to convert complex behaviors of external formats into IRIS Performer, it is still very straight forward to migrate basic structure, geometry, and graphics state into efficient IRIS Performer run-time structures via the functionality provided in the IRIS Performer database builder - pfdBuilder.

Scene Graph Creation using Nodes as defined in *libpf*

Creating an IRIS Performer scene graph requires a definite knowledge of the following IRIS Performer *libpf* node types - pfScene, pfGroup and pfGeode.

These nodes can be used to define a minimally functional IRIS Performer scene graph. See Chapter 5 for more details on *libpf* and IRIS Performer scene graphs and node types.

Defining Geometry and Graphics State for *libpr*

In order to input geometry and graphics into IRIS Performer, it is important to have an understanding of how IRIS Performer's low level rendering objects work in *libpr*, IRIS Performer's performance rendering library. The main libpr rendering primitives are a pfGeoSet and a pfGeoState. A

pfGeoSet is a collection of like geometric primitives that can all be rendered in exactly the same way in one large continuous chunk. A pfGeoState is a complete definition of graphics mode settings for the rendering hardware and software. It contains many attributes such as texture and material. Given a pfGeoSet and a corresponding pfGeoState, libpr can completely and efficiently render all of the geometry in the pfGeoSet. For a more detailed description of pfGeoSets and pfGeoStates see Chapter 10 which goes into detail on all libpr primitives and how IRIS Performer will use them.

However, realizing that IRIS Performer's structuring of geometry and graphics state is optimized for rendering speed and not for modelling ease or general conceptual partitioning, IRIS Performer now contains a new mechanism for translating external graphics state and geometry into efficient *libpr* structures. This new mechanism is the pfdBuilder that exists in *libpfdu*.

The pfdBuilder allows the immediate mode input of graphics state and primitives through very simple and exposed data structures. After having received all of the relevant information, the pfdBuilder builds efficient and somewhat optimized libpr data structures and returns a low-level libpf node that can be attached to an IRIS Performer scene graph. The pfdBuilder is the recommended method of importing data from non IRIS Performer-based formats into IRIS Performer.

Creation of a Performer Database Converter using *libpfdu*

Creating a new format converter is very simple process. More than thirty database loaders are shipped with IRIS Performer in source code form to serve as practical examples of this process. The loaders read formats that range from trivial to complex, and should serve as an instructive starting point for those developing loaders for other formats. These loaders can be found in the directory */usr/share/Performer/src/lib/libpfdb/libpf**.

This section describes the libpfdu framework for creating a 3-D database format converter. Let's consider writing a converter for a simple ASCII format that is called the Imaginary Immediate Mode format with the file type extension "*.iim*". This format is much like the more elaborate "*.im*" format loader used at SGI for the purposes of testing basic IRIS Performer functionality.

The first thing to do is set up the routine that **pfdLoadFile()** will call when it attempts to load a file with the extension ".iim".

```
extern pfNode *pfdLoadFile_iim(char *fileName)
{
}
```

This function needs to perform several basic actions:

1. Find and open the given file.
2. Reset the libpfdu pfdBuilder for input of new geometry and state.
3. Setup any pfdBuilder modes that the converter needs enabled.
4. Setup local data structures that can be used to communicate geometry and graphics state with the pfdBuilder.
5. Setup a libpf pfGroup which can hold all of the logical partitions of geometry in the file (or hold a subordinate collection of nodes as a general scene graph if the format supports it).
6. Optionally set up a default state to use for geometry with unspecified graphics state.
7. Parse the file which entails:
   - Filling in the local geometry and graphics state data structures.
   - Passing them to the pfdBuilder as inputted from the file
   - Ask the pfdBuilder to build the data structures into IRIS Performer data structures when a logical partition of the file has ended.
   - Attach the IRIS Performer node returned by the build to the higher level group which will hold the entire IRIS Performer representation of this file. Note that this step becomes more complex if the format supports the notion of hierarchy only in that the appropriate libpf nodes must be created and attached to each other via **pfAddChild()** to build the hierarchy. In this case requests are made for the builder to build after inputting all of the geometry and state found in a particular leaf node in the database.
8. Delete local data structures used to input geometry and graphics state.
9. Close the file.

10. Perform any optional optimization of the IRIS Performer scene graph. Optimizations might include calls to **pfdFreezeTransforms()**, **pfFlatten()** or **pfdCleanTree()**.
11. Return the pfGroup containing the entire IRIS Performer representation of the database file.

Steps 1-8 expand the function outline to the following:

```
extern pfNode *pfdLoadFile_iim(char *fileName)
{
  FILE* iimFile;
  pfdGeom* polygon;
  pfGroup* root;

  /* Performer has utility for finding and opening file */
  if ((iimFile = pfdOpenFile(fileName)) == NULL)
  return NULL;

  /* Clear builder from previous converter invocations */
  pfdResetBldrGeometry();
  pfdResetBldrState();

  /* Call pfdBldrMode for any needed modes here */

  /* Create polygon structure */
  /* holds one N-sided polygon where N is < 300 */
  polygon = pfdNewGeom(300);

  /* Create pfGroup to hold entire database */
  /* loaded from this file */
  root = pfNewGroup();

  /* Specify state for geometry with no graphics state */
  /* As well as default enables, etc. This routine */
  /* should invoke pfdCaptureDefaultBldrState()*/
  SetupDefaultGraphicsStateIfThereIsOne();

  /* Do all the real work in parsing the file and */
  /* converting into Performer */
  ParseIIMFile(iimFile, root, polygon);

  /* Delete local polygon struct */
  pfdDelGeom(polygon);
```

```
  /* Close File */
  fclose(iimFile);

  /* Optimize IRIS Performer scene graph */
  /* via use of pfFlatten, pfdCleanTree, etc. */
  OptimizeGraph(root);

  return (pfNode*)root;
}
```

Now, for at the heart of the file loader lies the **ParseIIMFile()** function. The specifics of parsing a file are completely dependent on the format so the parsing will be left as an exercise to the reader. However, the following code fragments should show a framework for what goes into integrating the parser with the pfdBuilder framework for geometry and graphics state data conversion. Note that several possible graphics state inheritance models might be used in external formats and that the pfdBuilder is designed to support all of them:

- The default pfdBuilder state inheritance is that of immediate mode graphics state. Immediate mode state is specified through calls to **pfdBldrStateMode()**, **pfdBldrStateAttr()**, and **pfdBldrStateVal()**.
- There also exists a pfdBuilder state stack for hierarchical state application to geometry. This is accomplished through the use of **pfdPushBldrState()** and **pfdPopBldrState()** in conjuncture with the normal use of the immediate mode pfdBuilder state API.
- Lastly, there is a pfdBuilder named state list that can be used to define a number of ‘named materials’ or ‘named state definitions’ that can then be recalled in one API called (for instance a user might define a ‘brick’ state with a red material and a brick texture. Later he might just want to say ‘brick’ is the current state and then input the walls of several buildings). This type of state naming is accomplished by fully specifying the state to be named via the immediate mode API, and then calling **pfdSaveBldrState()**. This state can then be recalled via **pfdLoadBldrState()**.

```
ParseIIMFile(FILE *iimFile, pfGroup *root, pfdGeom *poly)
{
  while((op = GetNextOp(iimFile)) != NULL)
  {
    switch(op)
    {
```

```
case GEOMETRY_POLYGON:
  polygon->numVerts = GetNumVerts(iimFile);

  /* Determine if polygon has Texture Coords */
  if (pfdGetBldrStateMode(PFSTATE_ENTEXTURE)==PF_ON)
    polygon->tbind = PFGS_PER_VERTEX;
  else
    polygon->tbind = PFGS_OFF;

  /* Determine if Polygon has normals */
  if (AreThereNormalsPerVertex() == TRUE)
    polygon->nbind = PFGS_PER_VERTEX;
  else if
    (pfdGetBldrStateMode(PFSTATE_ENLIGHTING)==PF_ON)
    polygon->nbind = PFGS_PER_PRIM;
  else
    polygon->nbind = PFGS_OFF;

  /* Determine if Polygon has colors */
  if (AreThereColorsPerVertex() == TRUE)
    polygon->cbind = PFGS_PER_VERTEX;
  else if (AreThereColorsPerPrim() == TRUE)
    polygon->cbind = PFGS_PER_PRIM;
  else
    polygon->cbind = PFGS_OFF;
  for(i=0;i<polygon->numVerts;i++)
  {
    /* Read ith Vertex into local data structure */
    polygon->coords[i][0] = GetNextVertexFloat();
    polygon->coords[i][1] = GetNextVertexFloat();
    polygon->coords[i][2] = GetNextVertexFloat();

    /* Read texture coord for ith vertex if any */
    if (polygon->tbind == PFGS_PER_VERTEX)
    {
      polygon->texCoords[i][0] = GetNextTexFloat();
      polygon->texCoords[i][1] = GetNextTexFloat();
    }

    /* Read normal for ith Vertex if normals bound*/
    if (polygon->nbind == PFGS_PER_VERTEX)
    {
      polygon->norms[i][0] = GetNextNormFloat();
      polygon->norms[i][1] = GetNextNormFloat();
      polygon->norms[i][2] = GetNextNormFloat();
```

```
    }
    /* Read only one normal per prim if necessary */
    else if ((polygon->nbind == PFGS_PER_PRIM) &&
             (i == 0))
    {
      polygon->norms[0][0] = GetNextNormFloat();
      polygon->norms[0][1] = GetNextNormFloat();
      polygon->norms[0][2] = GetNextNormFloat();
    }

    /* Get Color for the ith Vertex if color bound*/
       if (polygon->cbind == PFGS_PER_VERTEX)
       {
         polygon->colors[i][0] =
           GetNextColorFloat();
         polygon->colors[i][1] =
           GetNextColorFloat();
         polygon->colors[i][2] =
           GetNextColorFloat();
       }
       /* Get one color per prim if necessary */
       else if ((polygon->cbind == PFGS_PER_PRIM) &&
                (i == 0))
       {
         polygon->colors[0][0] =
           GetNextColorFloat();
         polygon->colors[0][1] =
           GetNextColorFloat();
         polygon->colors[0][2] =
           GetNextColorFloat();
       }
     }
     /* Add this polygon to pfdBuilder */
     /* Because it is a single poly, 1 */
     /* is specified here */
     pfdAddBldrGeom(1);
     break;
   case GRAPHICS_STATE_TEXTURE:
     {
       char *texName;
       pfTexture *tex;
       texName = ReadTextureName(iimFile);
       if (texName != NULL)
       {
         /* Get prototype tex from pfdBuilder*/
```

```
                    tex =
                      pfdGetTemplateObject(pfGetTexClassType())
;

                    /* This clears that object to default */
                    pfdResetObject(tex);

                    /* If just the name of a pfTexture is */
                    /* set, pfdBuilder will auto find & Load */
                    /* the texture*/
                    pfTexName(tex,texName);

                    /* This is the current pfdBuilder */
                    /* texture and texturing is on */
                    pfdBldrStateAttr(PFSTATE_TEXTURE,tex);
                    pfdBldrStateMode(PFSTATE_ENTEXTURE, PF_ON);
                  }
                  else
                  {
                    /* No texture means disable texturing */
                    /* And set current texture to NULL */
                    pfdBldrStateMode(PFSTATE_ENTEXTURE,PF_OFF);
                    pfdBldrStateAttr(PFSTATE_TEXTURE, NULL);
                  }
                 }
                 break;
            case GRAPHICS_STATE_MATERIAL:
              {
                pfMaterial *mtl;
                mtl = pfdGetTemplateObject(pfGetMtlClassType());
                pfdResetObject(mtl);
                pfMtlColor(mtl, PFMTL_AMBIENT,
                  GetAmRed(), GetAmGreen(), GetAmBlue());
                pfMtlColor(mtl, PFMTL_DIFFUSE,
                  GetDfRed(), GetDfGreen(), GetDfBlue());
                pfMtlColor(mtl, PFMTL_SPECULAR,
                  GetSpRed(), GetSpGreen(), GetSpBlue());
                pfMtlShininess(mtl, GetMtlShininess());
                pfMtlAlpha(mtl, GetMtlAlpha());
                pfdBldrStateAttr(PFSTATE_FRONTMTL, mtl);
                pfdBldrStateAttr(PFSTATE_BACKMTL, mtl);
              }
              break;
            case GRAPHICS_STATE_STORE:
              pfdSaveBldrState(GetStateName());
```

```
            break;
          case GRAPHICS_STATE_LOAD:
            pfdLoadBldrState(GetStateName());
            break;
          case GRAPHICS_STATE_PUSH:
            pfdPushBldrState();
            break;
          case GRAPHICS_STATE_POP:
            pfdPopBldrState();
            break;
          case GRAPHICS_STATE_RESET:
            pfdResetBldrState();
            break;
          case GRAPHICS_STATE_CAPTURE_DEFAULT:
            pfdCaptureDefaultBldrState();
            break;
          case BEGIN_LEAF_NODE:
            /* Not really necessary because it is */
            /* destroyed on build*/
            pfdResetBldrGeometry();
            break;
          case END_LEAF_NODE:
               {
                 pfNode *nd = pfdBuild();
                 if (nd != NULL)
                   pfAddChild(root,nd);
               }
               break;
           }
         }
       }
```

One of the fundamental structures involved in the above routine outline is the pfdGeom structure which users fill in with information about a single primitive, or a single strip of primitives. The pfdGeom structure is essential in communicating with the pfdBuilder and is defined as follows:

```
typedef struct _pfdGeom
    {
     int flags;
     int nbind, cbind, tbind;

     int numVerts;
     short primtype;
```

```
    float pixelsize;

    /* Non-indexed attributes */
    /* ..do not set if poly is indexed */
    pfVec3 *coords;
    pfVec3 *norms;
    pfVec4 *colors;
    pfVec2 *texCoords;

    /* Indexed attributes */
    /* ..do not set if poly is non-indexed */
    pfVec3 *coordList;
    pfVec3 *normList;
    pfVec4 *colorList;
    pfVec2 *texCoordList;

    /* Index lists*/
    /* ..do not set if poly is non-indexed */
    ushort *icoords;
    ushort *inorms;
    ushort *icolors;
    ushort *itexCoords;

    struct _pfdGeom *next;
   } pfdGeom;
```

See the pfdGeoBuilder(3pf) reference pages for more information on using this structure along with its sister structure, the pfdPrim.

The above should provide a well-defined framework for creating a database converter that can be used with any IRIS Performer applications via the **pfdLoadFile()** functionality.

However, it is also important to note that there are a multitude of pfdBuilder modes and attributes that can be used to affect some of the basic methods that the builder actually uses:

Table 9-5 pfdBuilder Modes and Attributes

Function Name	Token Description
pfd{Get}BldrMode	**PFDBLDR_MESH_ENABLE**
	PFDBLDR_MESH_SHOW_TSTRIPS
	PFDBLDR_MESH_INDEXED
	PFDBLDR_MESH_MAX_TRIS
	PFDBLDR_MESH_RETESSELLATE
	PFDBLDR_MESH_LOCAL_LIGHTING
	PFDBLDR_AUTO_COLORS
	PFDBLDR_AUTO_NORMALS
	PFDBLDR_AUTO_ORIENT
	PFDBLDR_AUTO_ENABLES
	PFDBLDR_AUTO_CMODE
	PFDBLDR_AUTO_DISABLE_TCOORDS_BY_STATE
	PFDBLDR_AUTO_DISABLE_NCOORDS_BY_STATE
	PFDBLDR_AUTO_LIGHTING_STATE_BY_NCOORDS
	PFDBLDR_AUTO_LIGHTING_STATE_BY_MATERIALS
	PFDBLDR_AUTO_TEXTURE_STATE_BY_TEXTURES
	PFDBLDR_AUTO_TEXTURE_STATE_BY_TCOORDS
	PFDBLDR_BREAKUP
	PFDBLDR_BREAKUP_SIZE
	PFDBLDR_BREAKUP_BRANCH
	PFDBLDR_BREAKUP_STRIP_LENGTH
	PFDBLDR_SHARE_MASK
	PFDBLDR_ATTACH_NODE_NAMES
	PFDBLDR_DESTROY_DATA_UPON_BUILD
	PFDBLDR_PF12_STATE_COMPATIBLE
	PFDBLDR_BUILD_LIMIT
	PFDBLDR_GEN_OPENGL_CLAMPED_TEXTURE_COORDS
	PFDBLDR_OPTIMIZE_COUNTS_NULL_ATTRS
pfd{Get}BldrAttr	**PFDBLDR_NODE_NAME_COMPARE**
	PFDBLDR_STATE_NAME_COMPARE

Because the pfdBuilder is released as source code, it is easy to add further functionality and more modes and attributes to even further customize this central functionality.

In fact, because the pfdBuilder acts as a “data funnel” in converting data into IRIS Performer run-time structures, it is easy to control the behavior of many standard conversion tasks through merely globally setting builder modes which will subsequently affect all converters that use the pfdBuilder to process their data.

Supported Database Formats

Vendors of several leading database construction and processing tools have provided database-loading software for you to use with IRIS Performer. This section describes these loaders, the loaders developed by the IRIS Performer engineering team, and several loaders developed in the IRIS Performer user community for other database formats.

Importing your databases is simple if they’re in formats for which IRIS Performer database loaders have already been written. Each of the loaders listed in Table 9-6 is included with IRIS Performer. If you want to import or export databases in any of these formats, refer to the appropriate section of this chapter for specific details about the individual loaders.

Table 9-6 Supported Database Formats

Name	Description
3ds	AutoDesk 3DStudio binary data
bin	SGI format used by powerflip
bpoly	Side Effects Software PRISMS binary data
byu	Brigham Young University CAD/FEA data
dwb	Coryphaeus Software Designer’s Workbench data
dxf	AutoDesk AutoCAD ASCII format
flt11	MultiGen public domain Flight v11 format
flt14	MultiGen OpenFlight v14 format

Table 9-6 (continued) Supported Database Formats

Name	Description
gds	McDonnell-Douglas GDS things data
gfo	Old SGI radiosity data format
im	Simple IRIS Performer data format
irtp	AAI/Graphicon Interactive Real-Time PHIGS
iv	SGI Open Inventor format (VRML 1.0 superset)
lsa	Lightscape Technologies ASCII radiosity data
lsb	Lightscape Technologies binary radiosity data
medit	Medit Productions medit modeling data
nff	Eric Haines' ray tracing test data
obj	Wavefront Technologies data format
pegg	Radiosity research data format
phd	SGI polyhedron data format
poly	Side Effects Software PRISMS ASCII data
ptu	Simple IRIS Performer terrain data format
s1k	US ARMY SIMNET database format
sgf	US Naval Academy standard graphics format
sgo	Paul Haeberli's graphics data format
spf	US Naval Academy simple polygon format
sponge	Sierpinski sponge 3D fractal generator
star	Astronomical data from Yale University star chart
stla	3D Structures ASCII stereolithography data
stlb	3D Structures binary stereolithography data
stm	Michael Garland's terrain data format
sv	John Kichury's i3dm modeler format

Table 9-6 (continued) Supported Database Formats

Name	Description
tri	University of Minnesota Geometry Center data
unc	University of North Carolina walkthrough data

Description of Supported Formats

AutoDesk 3DS Format

The AutoDesk 3DS format is used by the 3DStudio program and by a number of 3D file-interchange tools. The IRIS Performer loader for 3DS files is located in the */usr/share/Performer/src/lib/libpfdb/libpf3ds* directory. This loader uses an auxiliary library, *3dsftk.a*, to parse and interpret the 3ds file.

pfdLoadFile() uses the function **pfdLoadFile_3ds()** to import data from 3DStudio files into IRIS Performer run-time data structures:

Silicon Graphics BIN Format

The Silicon Graphics BIN format is supported by both Showcase™ and the *powerflip* demonstration program. BIN files are in a simple format that specifies only independent quadrilaterals.

The image in Figure 9-1 shows several of the BIN-format objects provided in the IRIS Performer sample data directory.

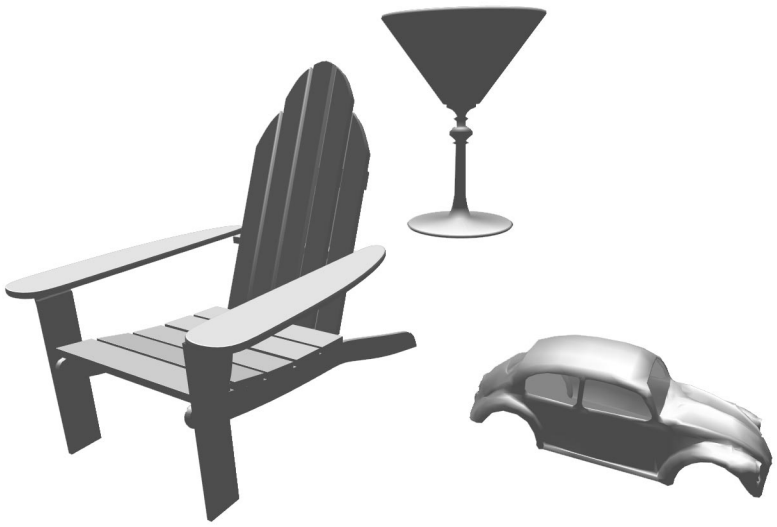


Figure 9-1 BIN-Format Data Objects

The source code for the BIN-format importer **pfdLoadFile_bin()** is provided in the file *pfbin.c*. This code shows how easy it can be to implement an importer. Since **pfdLoadFile_bin()** is based on the **pfdBuilder()** utility function, it will build efficient triangle-strip pfGeoSets from the quadrilaterals of a given BIN file. The BIN format has the following structure:

1. A 4-byte magic number, 0x5432, which identifies the file as a BIN file.
2. A 4-byte number that contains the number of vertices, which is four times the number of quadrilaterals.
3. Four bytes of zero.
4. A list of polygon data for each vertex in the object. The data consists of three floating-point words of information about normals, followed by three floating-point words of vertex information.

The BIN format uses these data structures:

```
typedef struct
{
    float normal[3];
    float coordinate[3];
} Vertex;

typedef struct
{
    long magic;
    long vertices;
    long zero;
    Vertex vertex[1];
} BinFile;
```

pfdLoadFile() uses the function **pfdLoadFile_bin()** to import data from BIN format files into IRIS Performer run-time data structures:

The pfdLoadFile_bin() function composes a random color for each file it reads. The chosen color has red, green, and blue components uniformly distributed within the range 0.2 to 0.7 and is fully opaque.

Side Effects POLY Format

The Side Effects software PRISMS database modeler format supports both ASCII and binary forms of the POLY format. The IRIS Performer loader for ASCII ".poly" files is located in the */usr/share/Performer/src/lib/libpfdb/libpfpoly* directory. The binary format ".bpoly" loader is located in the directory */usr/share/Performer/src/lib/libpfdb/libpfbpoly*. These formats are equivalent in content and differ only in representation.

The POLY format is an easy to understand ASCII data representation with the following structure:

1. A text line containing the keyword "POINTS"
2. One text line for each vertex in the file. Each line begins with a vertex number, followed by a colon, followed by the X, Y, and Z axis coordinates of the vertex, optional additional information, and a new-line character. The optional information includes color

specification in the form “c(R,G,B,A)”, a normal vector of the form “n(NX,NY,NZ)”, or a texture coordinate in the form “uv(S,T)” where each of the values shown are floating point numbers.

3. A text line containing the keyword “POLYS”

4. One text line for each polygon in the file. Each line begins with a polygon number, followed by a colon, followed by a series of vertex indices, optional additional information, an optional “<“ character, and a new-line. The optional information includes color specification in the form “c(R,G,B,A)”, a normal vector of the form “n(NX,NY,NZ)”, or a texture coordinate in the form “uv(S,T)” where the values in parentheses are floating point numbers.

Here is a sample POLY format file for a cube with colors, texture coordinates, and normals specified at each vertex:

```
POINTS
1: -0.5 -0.5 -0.5 c(0, 0, 0, 1) uv(0, 0) n(0, -1, 0)
2: -0.5 -0.5 0.5 c(0, 0, 1, 1) uv(0, 0) n(0, -1, 0)
3: 0.5 -0.5 0.5 c(1, 0, 1, 1) uv(1, 0) n(0, -1, 0)
4: 0.5 -0.5 -0.5 c(1, 0, 0, 1) uv(1, 0) n(0, -1, 0)
5: -0.5 -0.5 0.5 c(0, 0, 1, 1) uv(0, 0) n(0, 0, 1)
6: -0.5 0.5 0.5 c(0, 1, 1, 1) uv(0, 1) n(0, 0, 1)
7: 0.5 0.5 0.5 c(1, 1, 1, 1) uv(1, 1) n(0, 0, 1)
8: 0.5 -0.5 0.5 c(1, 0, 1, 1) uv(1, 0) n(0, 0, 1)
9: -0.5 0.5 0.5 c(0, 1, 1, 1) uv(0, 1) n(0, 1, 0)
10: -0.5 0.5 -0.5 c(0, 1, 0, 1) uv(0, 1) n(0, 1, 0)
11: 0.5 0.5 -0.5 c(1, 1, 0, 1) uv(1, 1) n(0, 1, 0)
12: 0.5 0.5 0.5 c(1, 1, 1, 1) uv(1, 1) n(0, 1, 0)
13: -0.5 -0.5 -0.5 c(0, 0, 0, 1) uv(0, 0) n(0, 0, -1)
14: 0.5 -0.5 -0.5 c(1, 0, 0, 1) uv(1, 0) n(0, 0, -1)
15: 0.5 0.5 -0.5 c(1, 1, 0, 1) uv(1, 1) n(0, 0, -1)
16: -0.5 0.5 -0.5 c(0, 1, 0, 1) uv(0, 1) n(0, 0, -1)
17: -0.5 -0.5 -0.5 c(0, 0, 0, 1) uv(0, 0) n(-1, 0, 0)
18: -0.5 0.5 -0.5 c(0, 1, 0, 1) uv(0, 1) n(-1, 0, 0)
19: -0.5 0.5 0.5 c(0, 1, 1, 1) uv(0, 1) n(-1, 0, 0)
20: -0.5 -0.5 0.5 c(0, 0, 1, 1) uv(0, 0) n(-1, 0, 0)
21: 0.5 0.5 0.5 c(1, 1, 1, 1) uv(1, 1) n(1, 0, 0)
22: 0.5 0.5 -0.5 c(1, 1, 0, 1) uv(1, 1) n(1, 0, 0)
23: 0.5 -0.5 -0.5 c(1, 0, 0, 1) uv(1, 0) n(1, 0, 0)
24: 0.5 -0.5 0.5 c(1, 0, 1, 1) uv(1, 0) n(1, 0, 0)
POLYS
1: 1 2 3 4 <
2: 5 6 7 8 <
```

```
3: 9 10 11 12 <
4: 13 14 15 16 <
5: 17 18 19 20 <
6: 21 22 23 24 <
```

pfdLoadFile() uses the functions **pfdLoadFile_poly()** and **pfdLoadFile_bpoly()** to import data from ".poly" and ".bpoly" format files into IRIS Performer run-time data structures:

Brigham Young University BYU Format

The Brigham Young University ".byu" format is used as an interchange format by some finite element analysis packages. The IRIS Performer loader for ".byu" files is located in the */usr/share/Performer/src/lib/libpfdb/libpfbyu* directory.

The format of a BYU file consists of four parts as defined below:

1. A text line containing four counts: the number of *parts*, the number of *vertices*, the number of *polygons*, and the number of *elements* in the connectivity array.
2. The part definition list, containing the starting polygon number and ending polygon number (one pair per line) for *parts* lines.
3. The vertex list, which has the X, Y, Z coordinates of each vertex in the database packed two per line. This means that vertices 1 and 2 are on the first line, 3 and 4 are on the second, and so on for (*vertices* + 1)/2 lines of text in the file.
4. The connectivity array, with an entry for each polygon. These entries may span multiple lines in the input file and each consists of three or more vertex indices with the last negated as an end of list flag. For example, if the first polygon were a quad, the connectivity array might start with "1 2 3 -4" to define a polygon that connects the first four vertices in order.

The following BYU format file defines two adjoining quads:

```
2 6 2 0
1 1
2 2
0 0 0 10 0 0
10 10 0 0 10 0
10 10 10 0 10 10
1 2 3 -4
4 3 5 -6
```

pfdLoadFile() uses the function **pfdLoadFile_byu()** to import data from ".byu" format files into IRIS Performer run-time data structures.

Designer's Workbench DWB Format

The binary DWB format is used for input and output by the Designer's Workbench, EasyT, and EasyScene database modeling tools produced by Coryphaeus Software. DWB is an advanced database format that directly represents many of IRIS Performer's attribute and hierarchical scene graph concepts.

An importer for this format, named **pfdLoadFile_dwb()**, has been provided by Coryphaeus Software for your use. The loader code and its associated documentation are in the */usr/share/Performer/src/lib/libpfdb/libpfdwb* directory.

The image in Figure 9-2 shows a model of the Soma Cube puzzle invented by Piet Hein. The model was created using Designer's Workbench. Each of the pieces is stored as an individual DWB-format file. Do you see how to form the 3 x 3 cube at the lower left from the seven individual pieces?

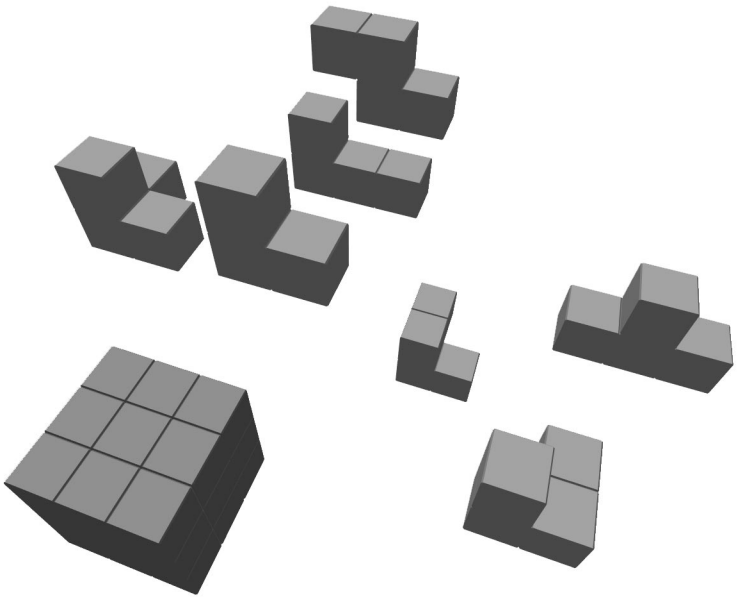


Figure 9-2 Soma Cube Puzzle in DWB Form

pfdLoadFile() uses the function **pfdLoadFile_dwb()** to load Designer's Workbench files into IRIS Performer run-time data structures.

AutoCAD DXF Format

The DXF format originated with Autodesk's AutoCAD database modeling system. The version recognized by the **pfdLoadFile_dxf()** database importer is a subset of ASCII Drawing Interchange Format (DXF) Release 12. The binary version of the DXF format, also known as DXF, isn't supported. Source code for the importer is in the file */usr/share/Performer/src/lib/libpfdb/libpfdxf/pfdxf.c*. **pfdLoadFile_dxf()** was derived from the DXF-to-DKB data file converter developed and placed in the public domain by Aaron A. Collins.

The image in Figure 9-3 shows a DXF model of the famous Utah teapot. This model was loaded from DXF format using the **pfdLoadFile_dxf()** database importer.

Figure 9-3 The Famous Teapot in DXF Form

The DXF format has an unusual though well-documented structure. The general organization of a DXF file is

1. HEADER section with general information about the file
2. TABLES section to provide definitions for named items, including:
   - LTYPE, the line-type table
   - LAYER, the layer table
   - STYLE, the text-style table
   - VIEW, the view table
   - UCS, the user coordinate-system table

- VPORT, the viewport configuration table
- DIMSTYLE, the dimension style table
- APPID, the application identification table

3. BLOCKS section containing block definition entities
4. ENTITIES section containing entities and block references
5. END-OF-FILE

Within each section are groups of values, where each value is defined by a two-line pair of tokens. The first token is a numeric code indicating how to interpret the information on the next line. For example, the sequence

```
10
1.000
20
5.000
30
3.000
```

defines a “start point” at the XYZ location (1, 5, 3). The codes 10, 20, and 30 indicate, respectively, that the primary X, Y, and Z values follow. All data values are retained in a set of numbered registers (10, 20, and 30 in this example), which allows values to be reused. This simple state-machine type of run-length coding makes DXF files space-efficient at the cost of making them harder to interpret.

pfdLoadFile() uses the function **pfdLoadFile_dxf()** to load DXF format files into IRIS Performer run-time data structures.

Several widely available technical books provide full details of this format if you need more information. Chief among these are *AutoCAD Programming, 2nd Edition*, by Dennis N. Jump, Windcrest Books, 1991, and *AutoCAD: The Complete Reference, Second Edition*, by Nelson Johnson, Osborne McGraw-Hill, 1991.

MultiGen OpenFlight Format

The OpenFlight format is a binary format used for input and output by the MultiGen and ModelGen database modeling tools produced by MultiGen. It is a comprehensive format that can represent nearly all of IRIS Performer's advanced concepts, including object hierarchy, instancing, level-of-detail selection, light-point specification, texture mapping, and material property specification.

MultiGen has provided an OpenFlight-format importer, **pfdLoadFile_flt()**, for your use. The loaders and associated documentation are in the directories */usr/share/Performer/src/lib/libpfdb/libpfflt11* and *libpfflt1*4. Refer to the *Readme* files in these directories for important information about the loaders and for help in contacting MultiGen for information about **pfdLoadFile_flt()** or the OpenFlight format.

The image in Figure 9-4 shows a model of a spacecraft created by Viewpoint Animation Engineering using MultiGen. This OpenFlight format model was loaded into IRIS Performer using **pfdLoadFile_flt()**.

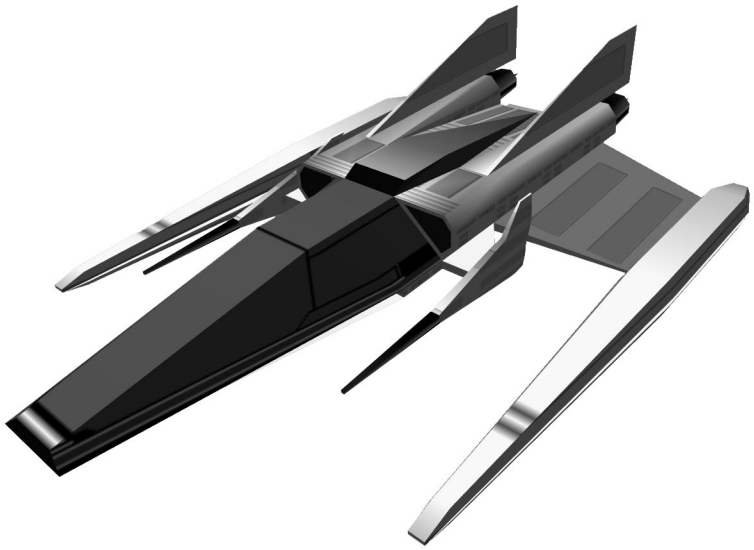


Figure 9-4 Spacecraft Model in FLIGHT Format

pfdLoadFile() uses the function **pfdLoadFile_flt()** to load OpenFlight format files into IRIS Performer run-time data structures.

Files in the OpenFlight format are structured as a linear sequence of records. The first few bytes of each record are a header containing an op-code, the length of the record, and possibly an ASCII name for the record. The first record in the file is a special "database header" record whose op-code, stored as a 2-byte short integer, has the value 1. This opcode header can be used to identify OpenFlight-format files. By convention, these files have a "*.flt*" filename extension.

pfdLoadFile_flt() makes use of several environment variables when locating data and texture files. These variables and several additional functions, including **pfdConverterMode_flt()**, **pfdGetConverterMode_flt()**, and **pfdConverterAttr_flt()** assist in OpenFlight file processing.

McDonnell-Douglas GDS Format

The ".gds" format (also known as the "Things" format) is used in at least one CAD system, and a minimal loader for this format has been developed for IRIS Performer users. The IRIS Performer loader for ".gds" files is located in the */usr/share/Performer/src/lib/libpfdb/libpfgds* directory.

The GDS format subset accepted by the **pfdLoadFile_gds()** function is easy to describe. It consists of the following five sequential sections in an ASCII file.

1. The number of *vertices*, which is given following a "YIN" tag.
2. The vertices, with one X, Y, Z triple per line for *vertices* lines.
3. The number zero on a line by itself.
4. The number of *polygons* on a line by itself.
5. A series of polygon definitions, each of which is represented on two or more lines. The first line contains the number one and the name of a material to use for the polygon. The next line or lines contain the indices for the polygons vertices. The first number on the first line is the number of *vertices*. This is followed by that number of vertex indices on that, and possibly subsequent, lines.

pfdLoadFile() uses the function **pfdLoadFile_gds()** to load ".gds" format files into IRIS Performer.

Silicon Graphics GFO Format

The GFO format is the simple ASCII format of the *barcelona* database that is provided in the IRIS Performer sample database directory. This database represents the famous German Pavilion at the Barcelona Exhibition of 1929, which was designed by Ludwig Mies van der Rohe and is shown in Figure 9-5.

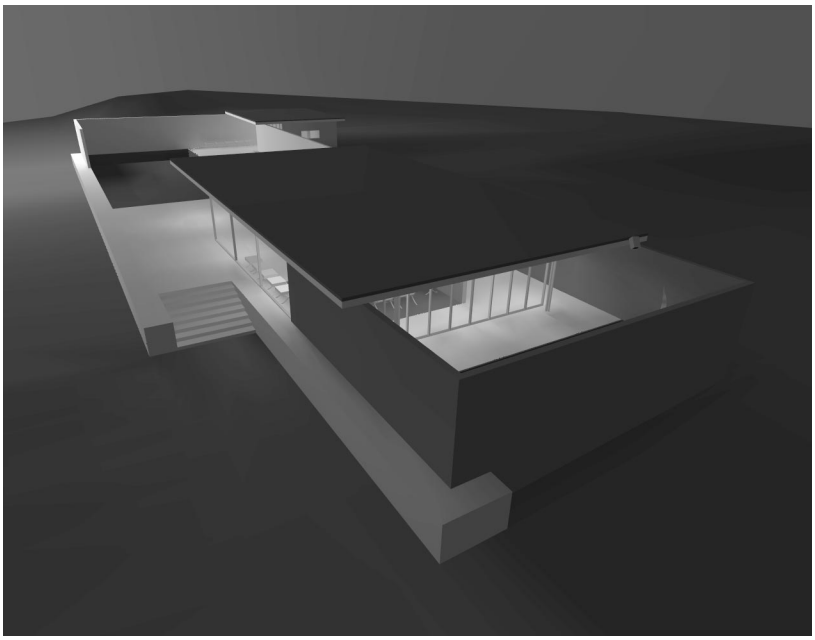


Figure 9-5 GFO Database of Mies van der Rohe's German Pavilion

The source code for the GFO-format loader is provided in the file */usr/share/Performer/src/lib/libpfdb/libpfgfo/pfbin.c*.

pfdLoadFile() uses the function **pfdLoadFile_gfo()** to load GFO format files into IRIS Performer run-time data-structures.

When working with GFO files, remember that hardware lighting isn't used since all illumination effects have already been accounted for with the ambient color at each vertex.

The GFO format defines polygons with a color at every vertex. It is the output format of an early radiosity system. Files in this format have a simple ASCII structure, as indicated by the following abbreviated GFO file:

```
scope {
v3f {42.9632 8.7500 0.9374}
cpack {0x8785a9}
v3f {42.9632 8.0000 0.9374}
```

```
cpack {0x8785a9}
...
v3f {-1.0000 -6.5858 10.0000}
cpack {0xffffff}
polygon {cpack[0] v3f[0] cpack[1] v3f[1] cpack[2] v3f[2] cpack[3] v3f[3] }
polygon {cpack[4] v3f[4] cpack[5] v3f[5] cpack[6] v3f[6] cpack[7] v3f[7] }
...
polygon {cpack[7330] v3f[7330] cpack[7331] v3f[7331] cpack[7332] v3f[7332]
cpack[7333] v3f[7333] }
instance {
polygon[0]
polygon[1]
...
polygon[2675]
}
}
```

This example is taken from the file *barcelona-l.gfo*, one of only two known databases in the GFO format. The importer uses functions from the *libpfdu* library (such as those from the pfdBuilder) to generate efficient shared triangle strips. This increases the speed with which GFO databases can be drawn and reduces the size and complexity of the loader, since the builder's functions hide the details of the pfGeoSet construction process.

Silicon Graphics IM Format

The ".im" format is a simple format developed for test purposes by the IRIS Performer engineering team. As new features are added to IRIS Performer, the ".im" loader is extended to allow experimentation and testing. A recent example of this is support for pfText, pfString, and pfFont objects which can be seen by running perfly on the sample data file *fontsample.im*. The IRIS Performer ".im" loader is in the */usr/share/Performer/src/lib/libpfdb/libpfim* directory.

Here is an example IM format file that creates an extruded 3D text string. Copy this to a file ending in the extension ".im" and load it into Perfly. For a complete example of how text is handled in IRIS Performer, use Perfly to examine the file */usr/share/Performer/data/fontsample2.im*.

```
breakup 0 0.0 0 0
new root top
end_root
```

```
new font mistr-extruded Mistr 3
end_font

new str_text textnode mistr-extruded 1
Hello World||
end_text

attach top textnode
```

pfdLoadFile() uses the function **pfdLoadFile_im()** to load "*.im*" format files into IRIS Performer run-time data structures:

pfdLoadFile_im() searches the current IRIS Performer file path for the named file and returns a pointer to the pfNode parenting the imported scene graph, or NULL if the file isn't readable or doesn't contain a valid database.

AAI/Graphicon IRTP Format

The AAI/Graphicon ".irtp" format is used by the TopGen database modeling system and by the Graphicon-2000 image generator. The name IRTP is an acronym for Interactive Real-Time PHIGS. The IRIS Performer ".irtp" loader is in the */usr/share/Performer/src/lib/libpfdb/libpfirtp* directory. Though loader does not support the more arcane IRTP features, such as binary separating planes or a global matrix table, it has served as a basis for porting applications to IRIS Performer and the RealityEngine.

pfdLoadFile() uses the function **pfdLoadFile_irtp()** to load IRTP format files into IRIS Performer run-time data-structures.

Silicon Graphics Open Inventor Format

The Open Inventor object-oriented 3D-graphics toolkit defines a persistent data format that is also a superset of the VRML networked graphics data format. The image in Figure 9-6 shows a sample Open Inventor data file.

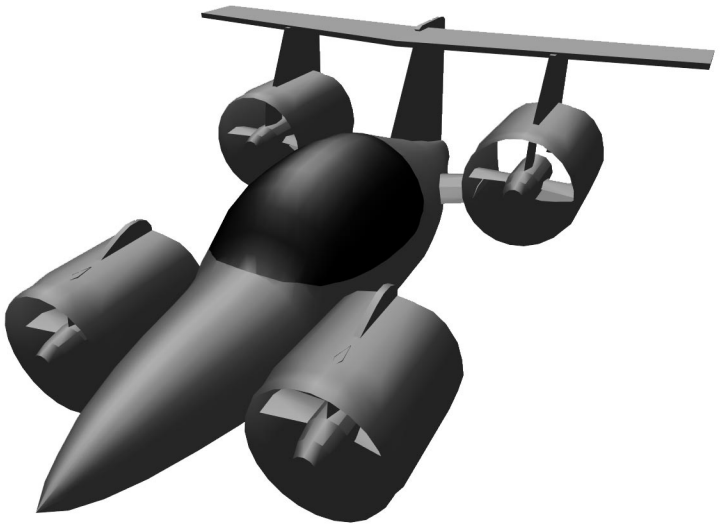


Figure 9-6 Aircar Database in IRIS Inventor Format

The model in Figure 9-6 represents one design for the perennial “personal aircar of the future” concept. It was created, using Imagine, by Mike Halvorson of Impulse, and was modeled after the Moller 400 as described in *Popular Mechanics*.

The Open Inventor data-file loader provided with IRIS Performer reads both binary and ASCII format Open Inventor data files. Open Inventor scene graph description files in both formats have the suffix “*.iv*” appended to their file names.

Here is a simple Open Inventor file that defines a cone:

```
#Inventor V2.1 ascii

Separator {
 Cone {
 }
}
```

The source code for the Open Inventor format importer is provided in the *libpfdb/libpfiv* source directory.

pfdLoadFile() uses the function **pfdLoadFile_iv()** to load Open Inventor format files into IRIS Performer run-time data-structures. Because VRML is a subset of Open Inventor 2.1, if you have Open Inventor 2.1 installed, you can also read VRML ".wrl" files using **pfdLoadFile()**. IRIS Performer also comes with an Inventor loader that works with Open Inventor 2.0, if Open Inventor 2.1 is not installed.

Lightscape Technologies LSA and LSB Formats

The Lightscape Visualization system is a product of Lightscape Technologies, Inc., and is designed to compute accurate simulations of global illumination within complex 3D environments. The output files created with Lightscape Visualization can be read into IRIS Performer for real-time visual exploration.

Lightscape Technologies provides importers for two of their database formats, the simple ASCII LSA format and the comprehensive binary LSB format. These loaders are in the */usr/share/Performer/src/lib/libpfdb/libpflsa* and *libpflsb* directories, in the files *pflsa.c* and *pflsb.c*.Files in the LSA format are in ASCII and have the following components:

1. a 4x4 view matrix representing a default transformation
2. counts of the number of independent triangles, independent quadrilaterals, triangle meshes, and quadrilateral meshes in the file
3. geometric data definitions

There are four types of geometric definitions in LSA files. The formats of these definitions are as shown in Table 9-7.

Table 9-7 Geometric Definitions in LSA Files

Geometric Type	Format
Triangle	t X1 Y1 Z1 C1 X2 Y2 Z2 C2 X3 Y3 Z3 C3
Triangle mesh	tm n X1 Y1 Z1 C1 X2 Y2 Z2 C2 ...
Quadrilateral	q X1 Y1 Z1 C1 X2 Y2 Z2 C2 X3 Y3 Z3 C3 X4 Y4 Z4 C4
Quadrilateral mesh	qm n X1 Y1 Z1 C1 X2 Y2 Z2 C2 ...

The C*n* values in Table 9-7 refer to colors in the format accepted by the IRIS GL function **cpack()**; these colors should be provided in decimal form. The X, Y, and Z values are vertex coordinates. Polygon vertex ordering in LSA files is consistently counter-clockwise, and polygon normals are not specified. The first few lines of the LSA sample file *chamber.0.lsa* provide an example of the format:

```
 0.486911  0.03228900  0.979046  0.9596590
-1.665110  0.00944197  0.286293  0.2806240
 0.000000  1.92730000 -0.017805 -0.0174524
 0.240398 -5.54670000 13.021200 13.4945000

1782 4751 0 0

t 4.35 -7.3677 2.57 6188666 6.5 -9.3 2.57 5663353 4.35 -9.3 2.57 5728890
t 6.5 -9.3 2.57 5663353 4.35 -7.3677 2.57 6188666 6.5 -8.2463 2.57 6057596
```

The count line indicates that the file contains 1782 independent triangles and 4751 independent quadrilaterals, which together represent 11,284 triangles. The image in Figure 9-7 shows this database, the New Jerusalem City Hall. This was produced by A. J. Diamond of Donald Schmitt and Company, Toronto, Canada, using the Lightscape Visualization system.

Figure 9-7 LSA-Format City Hall Database

pfdLoadFile() uses the function **pfdLoadFile_lsa()** to load LSA format files into IRIS Performer run-time data-structures.

Files in the LSB binary format have a very different structure from LSA files. Representing not just polygon data, they contain much of the structural information present in the "*.ls*" files used by the Lightscape Visualization system, including material, layer, and texture definitions as well as a hierarchical mesh definition for geometry. This information is structured as a series of data sections, which include:

- the signature, a text string that identifies the file
- the header, which contains global file information
- the material table, defining material properties

- the layer table, defining grouping and association
- the texture table, referencing texture images
- geometry in the form of clusters

The format of the geometric clusters is somewhat complicated. A cluster is a group of coplanar surfaces called patches that share a common material, layer, and normal. Each patch shares at least one edge with another patch in the cluster. Each patch defines either a convex quadrilateral or a triangle, and patches represent quad-trees called nodes. Each node points to its corner vertices and its children. The leaf nodes point to their corner vertices and the child pointers can optionally point to the vertices that split an edge of the node. Only the locations of vertices that are corners of the patches are stored in the file; other vertices are created by subdividing nodes of the quad-tree as the LSB file is loaded. The color information for each vertex is unique and is specified in the file.

The image in Figure 9-8 shows an LSB-format database developed during the design of a hospital operating room. This database was produced by the DeWolff Partnership of Rochester, New York, using the Lightscape Visualization system.

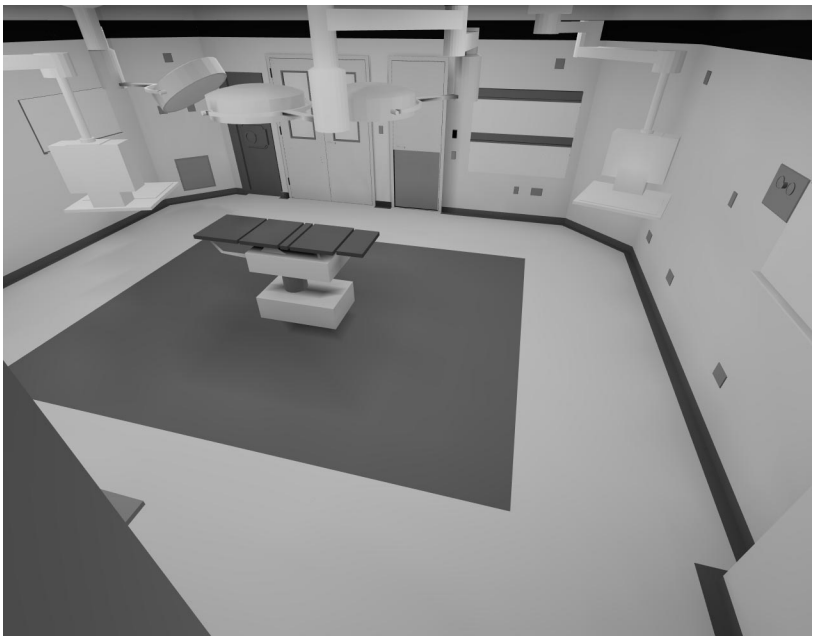


Figure 9-8 LSB-Format Operating Room Database

pfdLoadFile() uses the function **pfdLoadFile_lsb()** to load LSB format files into IRIS Performer run-time data-structures.

When working with Lightscape Technologies files, remember that hardware lighting isn't needed because all illumination effects have already been accounted for with the ambient color at each vertex.

Medit Productions MEDIT Format

The ".medit" format is used by the Medit database modeling system produced by Medit Productions. The Performer ".medit" loader is in the */usr/share/Performer/src/lib/libpfdb/libpfmedit* directory.

pfdLoadFile() uses the function **pfdLoadFile_medit()** to load MEDIT format files into IRIS Performer run-time data-structures.

NFF Neutral File Format

The ".nff" format was developed by Eric Haines as a way to provide standard procedural databases for evaluating ray tracing software. IRIS Performer includes an extended NFF loader with superquadric torus support, a named `build` keyword, and numerous small bug fixes. The ".nff" loader is located in the */usr/share/Performer/src/lib/libpfdb/libpfnff* directory.

The file */usr/share/Performer/data/sampler.nff* uses each of the NFF data types. It is an excellent way to explore the "Show Tree", "Draw Style", and "Highlight Mode" features of Perfly. It is included here:

```
#-- torus
f .75 .00 .25 .6 .8 20 0
t 5 5 0 0 0 1 2 1
build torus

#-- cylinder
f .00 .75 .25 .6 .8 20 0
c
15 5 -3 2
15 5 3 2
#-- put a disc on the top and bottom of the cylinder
d 15 5 -3 0 0 -1 0 2
d 15 5 3 0 0 1 0 2
build cylinder

#-- cone
f .00 .25 .75 .6 .8 20 0
c
25 5 -3 3
25 5 3 0
#-- put a disc on the bottom of the cone
d 25 5 -3 0 0 -1 0 3
build cone

#-- sphere
f .75 .00 .75 .6 .8 20 0
s 5 15 0 3
build sphere

#-- hexahedron
f .25 .25 .50 .6 .8 20 0
h 13 13 -2 17 17 2
```

```
build hexahedron

#-- superquadric sphere
f .80 .10 .30 .6 .8 20 0
ss 25 15 0 2 2 2 .1 .4
build superquadric_sphere

#-- disc (washer shape)
f .20 .20 .90 .6 .8 20 0
d 5 25 0 0 0 1 1 2.5
build disc

#-- grid (height field)
f .80 .80 .10 .6 .8 20 0
g 4 4 12 18 22 28 0 4
0 0 0 0
0 1 0 0
0 0 -1 0
0 0 0 0
build grid

#-- superquadric torid
f .40 .20 .60 .6 .8 20 0
st 25 25 0 0.5 0.5 0.5 .33 .33 3
build superquadric_torid

#-- polygon with no normals
f .20 .20 .20 .6 .8 20 0
p 4
-5 -5 -10
35 -5 -10
35 35 -10
-5 35 -10
build polygon
```

pfdLoadFile() uses the function **pfdLoadFile_nff()** to load NFF format files into IRIS Performer run-time data-structures.

Wavefront Technology OBJ Format

The OBJ format is an ASCII data representation read and written by the Wavefront Technology *Model* program. A number of database models in this format have been placed in the public domain, making this a useful format

to have available. IRIS Performer provides the function **pfdLoadFile_obj()** to import OBJ files. The source code for **pfdLoadFile_obj()** is in the file *pfobj.c* in the */usr/share/Performer/src/lib/libpfdb/libpfobj* loader source directory.

The OBJ-format database shown in Figure 9-9 models an office building that's part of the Silicon Graphics corporate campus in Mountain View, California.

Figure 9-9 Silicon Graphics Office Building as OBJ Database

Files in the OBJ format have a flexible all-ASCII structure, with simple keywords to direct the parsing of the data. This format is best illustrated with a short example that defines a texture-mapped square:

```
#-- 'v' defines a vertex; here are four vertices
v -5.000000  5.000000 0.000000
v -5.000000 -5.000000 0.000000
v  5.000000 -5.000000 0.000000
v  5.000000  5.000000 0.000000

#-- 'vt' defines a vertex texture coordinate; four are given
```

```
vt 0.000000 1.000000 0.000000
vt 0.000000 0.000000 0.000000
vt 1.000000 0.000000 0.000000
vt 1.000000 1.000000 0.000000

#-- ‘usemtl’ means select the material definition defined
#-- by the name MaterialName
usemtl MaterialName

#-- ‘usemap’ means select the texturing definition defined
#-- by the name TextureName
usemap TextureName

#-- ‘f’ defines a face. This face has four vertices ordered
#-- counter-clockwise from the upper left in both geometric
#-- and texture coordinates. Each pair of numbers separated
#-- by a slash indicates vertex and texture indices,
#-- respectively, for a polygon vertex.
f 1/1 2/2 3/3 4/4
```

pfdLoadFile() uses the function **pfdLoadFile_obj()** to load Wavefront OBJ files into IRIS Performer run-time data-structures.

Silicon Graphics PHD Format

The PHD format was created to describe the geometric polyhedron definitions derived mathematically by Andrew Hume and by the *Kaleido* program of Zvi Har’El. This format describes only the geometric shape of polyhedra; it provides no specification for color, texture, or appearance attributes such as specularity.

The IRIS Performer sample data directories contain numerous polyhedra in the PHD format. The image in Figure 9-10 shows many of the polyhedron definitions laboriously computed by Andrew Hume.

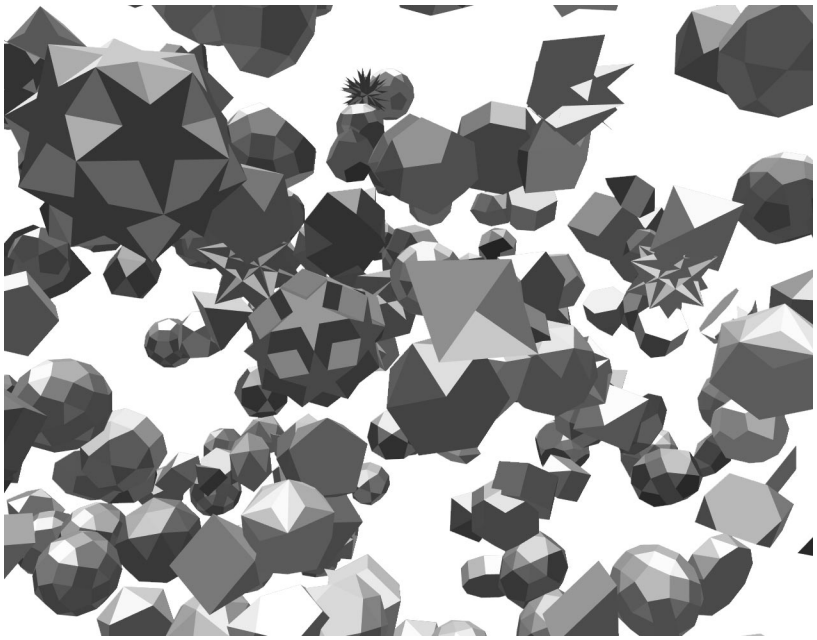


Figure 9-10 Plethora of Polyhedra in PHD Format

The source code for the PHD-format importer is in the file */usr/share/Performer/src/lib/libpfdb/libpfpoly/pfphd.c*.

PHD format files have a line-structured ASCII form; an initial keyword defines the contents of each line of data. The file format consists of a filename definition (introduced by the keyword *file*) followed by one or more object definitions.

Object definitions are bracketed by the keywords *object.begin* and *object.end* and contain one or more polygon definitions. Objects can have a name in quotes following the *object.begin* keyword; such a name is used by the loader for the name of the corresponding IRIS Performer node.

Polygon definitions are bracketed by the keywords *polygon.begin* and *polygon.end* and contain three or more vertex definitions.

Vertex definitions are introduced by the *vertex* keyword and define the X, Y, and Z coordinates of a single vertex.

The following is a PHD-format definition of a unit-radius tetrahedron centered at the origin of the coordinate axes. It is derived from the database developed by Andrew Hume but has since been translated, scaled, and reformatted.

```
file 000.phd
object.begin "tetrahedron"
polygon.begin
vertex -0.090722 -0.366647  0.925925
vertex  0.544331 -0.628540 -0.555555
vertex  0.453608  0.890430  0.037037
polygon.end
polygon.begin
vertex -0.907218  0.104757 -0.407407
vertex -0.090722 -0.366647  0.925925
vertex  0.453608  0.890430  0.037037
polygon.end
polygon.begin
vertex -0.090722 -0.366647  0.925925
vertex -0.907218  0.104757 -0.407407
vertex  0.544331 -0.628540 -0.555555
polygon.end
polygon.begin
vertex  0.453608  0.890430  0.037037
vertex  0.544331 -0.628540 -0.555555
vertex -0.907218  0.104757 -0.407407
polygon.end
object.end
```

pfdLoadFile() uses the function **pfdLoadFile_phd()** to load PHD format files into IRIS Performer run-time data-structures.

The pfdLoadFile_phd() function composes a color with red, green, and blue components uniformly distributed within the range 0.2 to 0.7 that is consistent for each polygon with the same number of vertices within a single polyhedron.

Silicon Graphics PTU Format

The PTU format is named for the *IRIS Performer Terrain Utilities*, of which the **pfdLoadFile_ptu()** function is the sole example at the present time. This function accepts as input the name of a control file (the file with the *".ptu"* filename extension) that defines the desired terrain parameters and references additional data files.

The database shown in Figure 9-11 represents a portion of the Yellowstone National Park. This terrain database was generated completely by the IRIS Performer Terrain Utility data generator from digital terrain elevation data and satellite photographic images. Image manipulation is performed using the Silicon Graphics ImageVision Library™ functions.

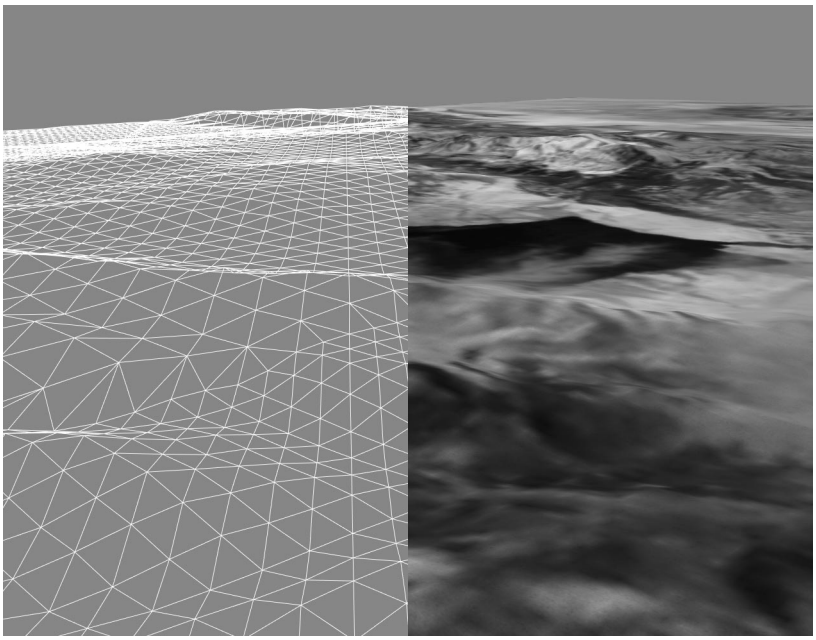


Figure 9-11 Terrain Database Generated by PTU Tools

The PTU control file has a fixed format that doesn't use keywords. The contents of this file are simply ASCII values representing the following data items:

1. The name to be assigned to the top-level pfNode built by **pfdLoadFile_ptu()**.
2. The number of desired levels-of-detail (LOD) for the resulting terrain surface. The **pfdLoadFile_ptu()** function will construct this many versions of the terrain, each representing the whole surface but with exponentially fewer numbers of polygons in each version.
3. The numbers of highest-LOD tiles that will tessellate the entire terrain surface in the X and Y axis directions.
4. Two numeric values that define the mapping of texture image pixels to world-coordinate terrain geometry. These values are the number of meters per *texel* (texture pixel) of filtered grid post data in the X and Y axis dimensions.
5. The name of an image file that represents terrain height at regularly spaced sample points in the form of a monochrome image whose brightness at each pixel indicates the height at that sample point. Additional arguments are the number of samples in the input image in the X and Y directions, as well as the desired number of samples in these directions. The **pfdLoadFile_ptu()** function resamples the grid posts from the original to the desired resolution by filtering the height image using SGI ImageVision Library functions.
6. The name of an image file that represents the terrain texture image at regularly spaced sample points. Subsequent arguments are the number of samples in the image in the X and Y directions as well as the desired number of samples in these directions. This image will be applied to the terrain geometry. The scale values provided in the PTU file allow the terrain grid and texture image to be adjusted to create an orthographic alignment.
7. An optional second texture-image filename that serves as a detail texture when the terrain is viewed on RealityEngine systems. This texture is used in addition to the base texture image.
8. An optional detail-texture spline-table definition. The blending of the primary texture image and the secondary detail texture is controlled by a blend table defined by this spline function. The spline table is optional even when a detail texture is specified. Detail texture and its associated blend functions are applicable only on RealityEngine systems.

The source code for the PTU-format importer is provided in the file */usr/share/Performer/src/lib/libpfdb/libpfptu/pfptu.c*.

pfdLoadFile() uses the function **pfdLoadFile_ptu()** to load PTU format files into IRIS Performer run-time data-structures.

SIMNET S1000 Format

The S1000 format is used by the United States ARMY SIMNET distributed simulation system for armored vehicle training. The IRIS Performer ".s1k" loader is in the */usr/share/Performer/src/lib/libpfdb/libpfs1k* directory. It consists of two libraries, an IRIS Performer loader library developed by Todd R. Pravata of Texas Instruments, Inc., in Plano Texas and an S1000 data format toolkit that may be requested from the following source:

Frank Abbruscato
Director of Data Support Division
Digital Processing Center
US Army Topographic Engineering Center
7701 Telegraph Road
Alexandria, VA 22310-3864
phone: 703-355-2954
email: fabbrusc@tec.army.mil

The Texas Instruments S1K Loader is built on top of the S1000 Data Base API developed by Loral Advanced Distributed Simulation. The S1000 API is used to access the following S1000 data base elements:

- 3-D polygons for terrain, micro-terrainand culture.
- Model instance data
- Model geometry
- Texture patterns
- General data base information

The following S1000 database structures are not accessed by the loader since they are two-dimensional representations of the database terrain and features:

- Pole sets
- Defragmented Terrain Areal Features

The S1K Loader does not currently support loading dynamic models (with articulated parts), though support is planned in future releases.

Control File

The S1000 data base loader uses the concept of a “control file” to control the database load. An S1K control file is an ascii file that substitutes for the formatted database file common to other loaders. The control file must have a “.s1k” file suffix to be recognized as an S1K control file. Keyword and parameter value pairs specified by the user are placed in the control file to tailor the database load to the user’s requirements. Examples of control file parameters are:

```
S1KPROJ <project_directory>
PROJECT <project_name>
ASSEMBLY_PREFIX <assembly_prefix>
SEARCH_WINDOW <xmin> <ymin> <xmax> <ymax>
ORIGIN <x> <y>
UTM
VIEWING_MODE [ OTW | THERMAL | BOTH ]
INCLUDE_INACTIVE_POLYGONS
MODEL_FILTER <filter_regexp>
VTX_NORMALS
LEAF_SIZE_MAX <integer_size>
```

Project selection

The minimum control file specifies which project to load. The default is to load the entire project. The assembly prefix is assumed to be the same as the project name unless the user has specified a different assembly prefix. The control file also allows the user to specify the S1K project directory. If the project directory is specified in the control file then this setting overrides the environment variable setting S1KPROJ if any. The user may load a portion of the entire S1000 data base by specifying the coordinates of a search

window (see the S1000 API User's Manual for details about search window selection).

Coordinate system specification

The database load occurs in either assembly (ASSY) coordinates or UTM. The S1000 data base origin (the ASSY origin) is at the southwest corner of the assembly. If UTM coordinates are selected then the search window is assumed to be given in UTM coordinates and the output will be in UTM coordinates. The user may however establish his own local reference frame by specifying an origin for the output database. Again, if UTM is specified, the origin is assumed to be given in UTM, otherwise ASSY. The S1000 API provides the capability to convert from UTM to assembly coordinates. The assembly coordinate frame is a topocentric rectangular grid located at the southwest corner of the assembly area.

Loader details

The loader creates a scene graph spatially organized as a quadtree. Each leaf node contains LAND and MICROTERRAIN polygons, NETWORK elements and other data. **pfdLoadFile()** uses the function **pfdLoadFile_s1k()** to load S1000 format files into IRIS Performer run-time data-structures.

USNA Standard Graphics Format

The ".sgf" format is used at the United States Naval Academy as a standard graphics format for geometric data. The loader was developed based on the description of the standard graphics format as described by David F. Rogers and J. Alan Adams in the book *Mathematical Elements for Computer Graphics.* The IRIS Performer ".sgf" format loader is located in the directory */usr/share/Performer/src/lib/libpfdb/libpfsgf*.

Here is the vector definition for four stacked squares in SGF form:

```
0, 0, 0
1, 0, 0
1, 1, 0
0, 1, 0
0, 0, 0
1.0e37, 1.0e37, 1.0e37
```

```
0, 0, 1
1, 0, 1
1, 1, 1
0, 1, 1
0, 0, 1
1.0e37, 1.0e37, 1.0e37
0, 0, 2
1, 0, 2
1, 1, 2
0, 1, 2
0, 0, 2
1.0e37, 1.0e37, 1.0e37
0, 0, 3
1, 0, 3
1, 1, 3
0, 1, 3
0, 0, 3
1.0e37, 1.0e37, 1.0e37
```

pfdLoadFile() uses the function **pfdLoadFile_sgf()** to load SGF format files into IRIS Performer run-time data-structures.

Silicon Graphics SGO Format

The Silicon Graphics Object format is used by several cool utility programs and was one of the first database formats supported by IRIS Performer. The image in Figure 9-12 shows a model generated by Paul Haeberli and loaded into *perfly* by the **pfdLoadFile_sgo()** database importer.

Figure 9-12 Model in SGO Format

Objects in the SGO format have per-vertex color specification and multiple data formats. Objects contained in SGO files are constructed from three data types:

- lists of quadrilaterals
- lists of triangles
- triangle meshes

Objects of different types can be included as data within one SGO file.

The SGO format has the following structure:

1. A magic number, 0x5424, which identifies the file as an SGO file.
2. A set of data for each object. Each object definition begins with an identifying token, followed by geometric data. There can be multiple object definitions in a single file. An end-of-data token terminates the file.

The layout of an SGO file is

```
<SGO-file magic number>
<data-type token for object #1>
<data for object #1>
<data-type token for object #2>
<data for object #2>
...
<data-type token for object #n>
<data for object #n>
<end-of-data token>
```

Each of the identifying tokens is 4 bytes long. Table 9-8 lists the symbol, value, and meaning for each token.

Table 9-8 Object Tokens in the SGO Format

Symbol	Value	Meaning
OBJ_QUADLIST	1	Independent quadrilaterals
OBJ_TRILIST	2	Independent triangles
OBJ_TRIMESH	3	Triangle mesh
OBJ_END	4	End-of-data token

The next word following any of the three object types is the number of 4-byte words of data for that object. The format of this data varies depending on the object type.

For quadrilateral list (OBJ_QUADLIST) and triangle list (OBJ_TRILIST) objects, there are nine words of floating-point data for each vertex, as follows:

1. Three words that specify the components of the normal vector at the vertex.
2. Three words that specify the red, green, and blue color components, scaled to the range 0.0 to 1.0.
3. Three words that specify the X, Y, and Z coordinates of the vertex itself.

In quadrilateral lists, vertices are in groups of four, so there are 4 X 9 = 36 words of data for each quadrilateral. In triangle lists, vertices are in groups of three, for 3 X 9 = 27 words per triangle.

The triangle mesh, OBJ_TRIMESH, is the most complicated of the three object data types. Triangle mesh data consists of a set of vertices followed by a set of mesh-control commands. Triangle mesh data has the following format:

1. A long word that contains the number of words in the complete triangle mesh data packet.
2. A long word that contains the number of floating-point words required by the vertex data, at nine words per vertex.
3. The data for each vertex, consisting of nine floating-point words representing normal, color, and coordinate data.
4. A list of triangle mesh controls.

The triangle mesh controls, each of which is one word in length, are listed in Table 9-9.

Table 9-9 Mesh Control Tokens in the SGO Format

Symbol	Value	Meaning
OP_BGNTMESH	1	Begin a triangle strip.
OP_SWAPTMESH	2	Exchange old vertices.
OP_ENDBGNTMESH	3	End, then begin a strip.
OP_ENDTMESH	4	Terminate triangle mesh.

The triangle-mesh controls are interpreted sequentially. The first control must always be OP_BGNTMESH, which initiates the mesh-decoding logic. After each mesh control is a word (of type long integer) that indicates how many vertex indices follow. The vertex indices are in byte offsets, so to access vertex *n*, you must use the byte offset n X 9 X 4. See the graphics library reference books listed under “Bibliography” on page xxix for more information on triangle meshes (particularly see the IRIS GL books, if you’re using IRIS GL, for information on the swap-triangle-mesh concept).

pfdLoadFile() uses the function **pfdLoadFile_sgo()** to load SGO format files into IRIS Performer run-time data-structures.

You can find the source code for the SGO-format importer in the file *pfsgo.c*. This importer doesn't attempt to decode any triangle meshes present in input files; instead, it terminates the file conversion process as soon as an OBJ_TRIMESH data-type token is encountered. If you use SGO-format files containing triangle meshes you'll need to extend the conversion support to include the triangle mesh data type.

USNA Simple Polygon File Format

The ".spf" format is used at the United States Naval Academy as a simple polygon file format for geometric data. The loader was developed based on the description in the book *Mathematical Elements for Computer Graphics*. The IRIS Performer ".spf" loader is in the */usr/share/Performer/src/lib/libpfdb/libpfspf* directory.

The following ".spf" format file is defined in that book.

```
polygon with a hole
14,2
4,4
4,26
20,26
28,18
28,4
21,4
21,8
10,8
10,4
10,12
10,20
17,20
21,16
21,12
9,1,2,3,4,5,6,7,8,9
5,10,11,12,13,14
```

If you look at this file in Perfly you will see that the hole is not cut out of the letter "A" as might be desired. Such computational geometry computations are not considered the province of simple database loaders.

pfdLoadFile() uses the function **pfdLoadFile_spf()** to load SPF format files into IRIS Performer run-time data-structures.

Sierpinski Sponge Loader

The Sierpinski Sponge (a.k.a. Menger Sponge) loader is not based on a data format but rather is a procedural data generator. The loader interprets the portion of the user provided "file name" before the period and extension as an integer which specifies the number of recursive subdivisions desired in data generation. For example, providing the pseudo filename "3.sponge" to perfly will result in the Sponge loader being invoked and generating a sponge object using three levels of recursion, resulting in a 35712 polygon database object. The IRIS Performer ".sponge" loader can be found in the */usr/share/Performer/src/lib/libpfdb/libpfsponge* directory.

pfdLoadFile() uses the function **pfdLoadFile_sponge()** to load Sponge format files into IRIS Performer run-time data-structures.

Star Chart Format

The ".star" format is a distillation of astronomical data from the Yale Compact Star Chart. The sample data file */usr/share/Performer/data/3010.star* contains data from the YCSC that has been reduced to a list of the 3010 brightest stars as seen from Earth and positioned as 3010 points of light on a unit-radius sphere. The IRIS Performer ".star" loader can read this data and is provided as a convenience for making dusk, dawn, and night-time scenes. The loader is in the */usr/share/Performer/src/lib/libpfdb/libpfstar* directory.

Data in a ".star" file is simply a series of ASCII lines with the "s" (for star) keyword followed by X, Y, and Z coordinates, brightness, and an optional name. Here are the 10 brightest stars (excluding Sol) in the ".star" format:

```
s -0.18746032  0.93921369 -0.28763914 1.00 Sirius
s -0.06323564  0.60291260 -0.79529721 1.00 Canopus
s -0.78377002 -0.52700269  0.32859191 1.00 Arcturus
s  0.18718566  0.73014212  0.65715599 1.00 Capella
s  0.12507832 -0.76942003  0.62637711 0.99 Vega
s  0.13051330  0.68228769  0.71933979 0.99 Capella
s  0.19507207  0.97036278 -0.14262892 0.98 Rigel
s -0.37387931 -0.31261155 -0.87320572 0.94 Rigil Kentaurus
```

```
s -0.41809806  0.90381104  0.09121194 0.94 Procyon
s  0.49255905  0.22369388 -0.84103900 0.92 Achernar
```

pfdLoadFile() uses the function **pfdLoadFile_star()** to load Star format files into IRIS Performer run-time data-structures.

3D Lithography STL Format

The STL format is used to define 3D solids to be imaged by 3D lithography systems. STL defines objects as collections of triangular facets, each with an associated face normal. The ASCII version of this format is known as STLA and has a very simple structure.

The image in Figure 9-13 shows a typical STLA mechanical CAD database. This model is defined in the *bendix.stla* sample data file.

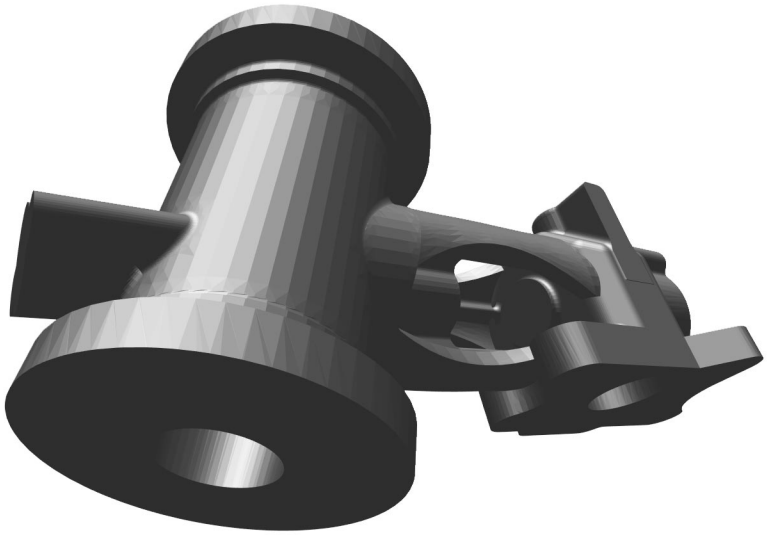


Figure 9-13 Sample STLA Database

The source code for the STLA-format loader is in the files */usr/share/Performer/src/lib/libpfdb/libpfstla/pfstla.c*.

STLA-format files have a line-structured ASCII form; an initial keyword defines the contents of each line of data. An STLA file consists of one or more facet definitions, each of which contains

1. the facet normal, indicated with the *facet normal* keyword
2. the facet vertices, bracketed by *outer loop* and *endloop* keywords
3. the *endloop* keyword

Here is an excerpt from *nut.stl*a, one of the STLA files provided in the IRIS Performer sample data directories. These are the first two polygons of a 524-triangle hex-nut object:

```
facet normal 0 -1 0
 outer loop
  vertex 0.180666 -7.62 2.70757
  vertex -4.78652 -7.62 1.76185
  vertex -4.436 -7.62 0
 endloop
endfacet
facet normal -0.381579 -0.921214 -0.075915
 outer loop
  vertex -4.48833 -7.59833 0
  vertex -4.436 -7.62 0
  vertex -4.78652 -7.62 1.76185
 endloop
endfacet
```

Use this function to import data from STLA-format files into IRIS Performer run-time data structures:

```
pfNode *pfdLoadFile_stla(char *fileName);
```

pfdLoadFile_stla() searches the current IRIS Performer file path for the file named by the *fileName* argument and returns a pointer to the pfNode that parents the imported scene graph, or NULL if the file isn't readable or doesn't contain recognizable STLA format data.

SuperViewer SV Format

The SuperViewer (SV) format is one of the several database formats that the I3DM database modeling tool can read and write. The I3DM modeler was developed by John Kichury of Silicon Graphics and is provided with IRIS Performer. The source code for the SV format importer is in the file *pfsv.c*.

The passenger vehicle database shown in Figure 9-14 was modeled using I3DM and is stored in the SV database format.

Figure 9-14 Early Automobile in SuperViewer SV Format

Within SV files, object geometry and attributes are described between text lines that contain the keywords *model* and *endmodel*. For example:

```
model wing
   geometry and attributes
endmodel
```

Any number of models can appear within a SuperViewer file. The geometry and attribute data mentioned above each consist of one of the following types:

- 3D Polygon with vertex normals and optional texture coordinates

  ```
  poly3dn <num_vertices> [textured]
  x1 y1 z1 nx1 ny1 nz1  [s1 t1]
  x2 y2 z2 nx2 ny2 nz2  [s2 t2]
  ...
  ```

  Where

  – *Xn Yn Zn* are the *n*th vertex coordinates

  – *Nxn Nyn Nzn* are the *n*th vertex normals

  – *Sn Tn* are the *n*th texture coordinates

- 3D Triangle mesh with vertex normals and optional texture coordinates

  ```
  tmeshn <num_vertices> [textured]
  x1 y1 z1 nx1 ny1 nz1  [s1 t1]
  x2 y2 z2 nx2 ny2 nz2  [s2 t2]
  ...
  ```

  Where

  – *Xn Yn Zn* are the *n*th vertex coordinates

  – *Nxn Nyn Nzn* are the *n*th vertex normals

  – *Sn Tn* are the *n*th texture coordinates

- Material definition. If the material directive exists before a model definition, it is taken as a new material specification. Its format is:

  ```
  material n Ar Ag Ab Dr Dg Db Sr Sg Sb Shine Er Eg Eb
  ```

  Where

  – *n* is an integer specifying a material number

  – *Ar Ag Ab* is the ambient color

  – *Dr Dg Db* is the diffuse color

  – *Sr Sg Sb* is the specular color

  – *Shine* is the material shininess

  – *Er Eg Eb* is the emissive color

If the material directive exists within a model description, the format is:

```
material n
```

Where *n* is an integer specifying which material (as defined by the material description above) is to be assigned to subsequent data.

- Texture definition. If the texture directive exists before a model definition it is taken as a new texture specification. Its format is:

  ```
  texture n TextureFileName
  ```

  If the texture directive exists within a model description, the format is:

  ```
  texture n
  ```

  Where *n* is an integer specifying which texture (as defined by the texture description above) is to be assigned to subsequent data.

- Backface polygon display mode. The backface directive is specified within model definitions to control backface polygon culling:

  ```
  backface mode
  ```

  Where a *mode* of “on” allows the display of backfacing polygons and a *mode* of “off” suppresses their display.

In actual use the SV format is somewhat self-documenting. Here is part of the SV file *apple.sv* from the */usr/share/Performer/data* directory:

```
material 20 0.0 0.0 0 0.400000 0.000000 0 0.333333 0.000000 0.0 10.0000 0 0 0
material 42 0.2 0.2 0 0.666667 0.666667 0 0.800000 0.800000 0.8 94.1606 0 0 0
material 44 0.0 0.2 0 0.000000 0.200000 0 0.000000 0.266667 0.0  5.0000 0 0 0

texture 4 prchmnt.rgb
texture 6 wood.rgb

model LEAF
material 44
texture 4
backface on
poly3dn 4 textured
 1.35265 1.35761 13.8338 0.0686595 -0.234553 -0.969676 0 1
 0.88243 0.96366 14.0329 0.0502096 -0.376701 -0.924973 0 0.75
-4.44467 1.24026 13.5669 0.0363863 -0.337291 -0.940697 0.0909091 0.75
-2.37938 2.17479 13.3626 0.0363863 -0.337291 -0.940697 0.0909091 1
poly3dn 4 textured
-2.37938 2.17479 13.3626 0.0363863 -0.337291 -0.940697 0.0909091 1
-4.44467 1.24026 13.5669 0.0363863 -0.337291 -0.940697 0.0909091 0.75
```

```
-9.23775 2.34664 13.1475 0.0344832 -0.284369 -0.958095 0.181818 0.75
-6.69592 3.94535 12.6716 0.0344832 -0.284369 -0.958095 0.181818 1
```

This excerpt specifies material properties and references texture images stored in the files *prchmnt.rgb* and *wood.rgb*, and then defines two polygons.

pfdLoadFile() uses the function **pfdLoadFile_sv()** to load SuperViewer files into IRIS Performer run-time data-structures.

Geometry Center Triangle Format

The ".tri" format is used at the University of Minnesota's Geometry Center as a simple geometric data representation. The loader was developed by inspection of a few sample files. The IRIS Performer ".tri" loader is in the */usr/share/Performer/src/lib/libpfdb/libpftri* directory.

These files have a very simple format: a line per vertex with position and normal given on each line as 6 ASCII numeric values. The file is simply a series of these triangle definitions. Here are the first two triangles from the data file */usr/share/Performer/data/mobrect.tri*:

```
1.788180  1.000870 0.135214 0.076169 -0.085488 0.993423
1.574000  0.925908 0.146652 0.089015 -0.086072 0.992304
1.793360  0.634711 0.099409 0.076402 -0.111845 0.990784
0.836848 -0.595230 0.197960 0.156677  0.044503 0.986647
0.709638 -0.345676 0.210010 0.157642  0.021968 0.987252
0.581200 -0.535321 0.234807 0.145068  0.030985 0.988936
```

pfdLoadFile() uses the function **pfdLoadFile_tri()** to load ".tri" format files into IRIS Performer run-time data-structures.

UNC Walkthrough Format

The ".unc" format was once used at the *University of North Carolina* as a format for geometric data in an architectural walkthrough application. The loader was developed based on inspection of a few sample files. The IRIS Performer ".unc" loader is in the */usr/share/Performer/src/lib/libpfdb/libpfunc* directory.

pfdLoadFile() uses the function **pfdLoadFile_unc()** to load UNC format files into IRIS Performer run-time data-structures.

Chapter 10

“libpr Basics”

This chapter discusses the rendering library that forms IRIS Performer’s foundation.

Chapter 10

libpr Basics

Earlier chapters of this guide described *libpf*, IRIS Performer's high-level visual simulation development library. Much of *libpf* is built on top of *libpr*, the high-performance rendering and utility library described in this chapter.

Overview

libpr provides many useful system- and hardware-oriented utilities as well as a high-performance interface to the graphics libraries. This interface eliminates the guesswork of the tuning process by providing optimized data structures and performance-tuned renderers.

The interface is intuitive, easy to learn, and flexible enough to let you plug any application into the hooks provided by *libpr* to access the full set of graphics library features. *libpr* is platform-independent, so you can realize performance gains across the entire Silicon Graphics product line.

Note: Both *libpr* and *libpf* object files are incorporated into a single library, called *libpf*, so the routines can be arranged in memory to improve caching behavior. However, *libpr* is still conceptually a stand-alone library and is referred to a such in the following discussion.

Design Motivation

A great deal of specialized knowledge is required to tune the performance of any piece of software. A primary design goal of IRIS Performer is to provide you with that knowledge in the form of an easy-to-use toolkit. The design decisions implemented in *libpr* combine an understanding of subtle machine-level implications with hands-on experience in tuning a variety of applications.

Applications that you develop using IRIS Performer will approach peak performance on both current and future hardware and software releases, so you don't have to spend a lot of time porting and tuning your code every time you make an upgrade to your system.

Key Features

libpr is a fundamental set of building blocks in much the same way that a graphics library is. It doesn't impose a multiprocessing orientation, nor is it targeted to any specific type of application; its only goal is optimizing the use of SGI graphics hardware. It can be used freely in conjunction with either the IRIS GL or the OpenGL graphics library. The library has four basic components:

- high-performance immediate-mode geometry rendering
- efficient management of Graphics Pipeline state
- fast and flexible analytic intersection-detection support
- common low-level programming utilities such as statistics gathering, window management, real-time clocks, and shared memory.

Some important features of *libpr* are that it

- encapsulates graphics library state in objects
- has a limited notion of hierarchy
- doesn't extend or manage all graphics library functions
- allows you to intermix graphics library and *libpr* calls freely
- provides functionality not available in lower-level graphics libraries
- is an easy porting target for applications

libpr provides highly optimized loops for rendering a wide variety of immediate-mode geometric primitives like points, lines, triangles, and triangle strips. Geometry appearance (is it textured, lit, transparent?) is determined by the current state of the Geometry Pipeline when the geometry is rendered. Efficient management of this state is crucial for best performance. To achieve the best performance, state changes must be minimized and rendering loops must be free of decisions. *libpr* employs two primary mechanisms for accomplishing this, pfGeoSet and pfGeoState.

Simply put, a pfGeoSet encapsulates 3D geometry while a pfGeoState encapsulates the appearance parameters known as *pipeline state*. A pfGeoSet is *drawn*—its vertices and other attributes are sent to the graphics library to be processed and ultimately rasterized on the screen. A pfGeoState is *applied*—its encapsulated state is used to configure the graphics library for a particular appearance, such as lit, textured, and fogged. A pfGeoSet can, and usually does, reference a pfGeoState thereby defining a both geometry and specific appearance attributes. When the pfGeoSet is then drawn, its associated pfGeoState is automatically applied first so that the geometry is rendered with the proper appearance.

Some specific *libpr* features and primitives discussed in this chapter are listed below:

- Fast rendering of geometry with a pfGeoSet,
- 3D Fonts with pfFont and pfString,
- Efficient state management with pfGeoStates and pfStates,
- Rendering effects with pfDecal for coplanar geometry, pfFog, pfLPointState, and pfHighlight,
- Texturing with pfTexture, pfTexEnv, and pfTexGen,
- Lighting with pfLight, pfLightModel, and pfMaterial,
- 3D transformations with pfSprites and pfMatrix,
- Flexible and efficient immediate mode display lists with pfDispList,
- Windowing with pfWindows,.
- Real-time clocks, and,
- Memory management with pfMemory, pfList, pfObject, pfDataPool, pfCycleMemory and pfCycleBuffer.

Geometry

All *libpr* geometry is defined by modular units that employ a flexible specification method. These basic groups of geometric primitives are termed pfGeoSets.

Geometry Sets

A pfGeoSet is a collection of geometry that shares certain characteristics. All items in a pfGeoSet must be of the same primitive type (whether they're points, lines, or triangles) and share the same set of attribute bindings (you can't specify colors-per-vertex for some items and colors-per-primitive for others in the same pfGeoSet). A pfGeoSet forms primitives out of lists of attributes that may be either indexed or nonindexed. An indexed pfGeoSet uses a list of unsigned short integers to index an attribute list. (See "Attributes" on page 325 for information about attributes and bindings.)

Indexing provides a more general mechanism for specifying geometry than hard-wired attribute lists and also has the potential for substantial memory savings as a result of shared attributes. Nonindexed pfGeoSets are sometimes easier to construct, usually a bit faster to render, and may save memory (since no extra space is needed for index lists) in situations where vertex sharing isn't possible. A pfGeoSet must be either completely indexed or completely nonindexed; it's not legal to have some attributes indexed and others nonindexed.

Note: *libpf* applications can include pfGeoSets in the scene graph with the pfGeode (Geometry Node).

Table 10-1 lists a subset of the routines that manipulate pfGeoSets.

Table 10-1 pfGeoSet Routines

Function	Description
pfNewGSet	Create a new pfGeoSet.
pfDelete	Delete a pfGeoSet.
pfCopy	Copy a pfGeoSet.
pfGSetGState	Specify the pfGeoState to be used.
pfGSetGStateIndex	Specify the pfGeoState index to be used.
pfGSetNumPrims	Specify the number of primitive items.
pfGSetPrimType	Specify the type of primitive.
pfGSetPrimLengths	Set the length of strip primitives.

Table 10-1 (continued) pfGeoSet Routines

Function	Description
pfGSetAttr	Set the attribute bindings.
pfGSetDrawMode	Specify draw mode, e.g., flat shading or wireframe.
pfGSetLineWidth	Set the line width for line primitives.
pfGSetPntSize	Set the point size for point primitives.
pfGSetHlight	Specify highlighting type for drawing.
pfDrawGSet	Draw a pfGeoSet.
pfGSetBBox	Specify a bounding box for the geometry.
pfGSetIsectMask	Specify an intersection mask for **pfGSetIsectSegs**.
pfGSetIsectSegs	Intersect line segments with pfGeoSet geometry.
pfQueryGSet	Determine the number of triangles or vertices.
pfPrint	Print the pfGeoSet contents.

Primitive Types

All primitives within a given pfGeoSet must be of the same type. To set the type of all primitives in a pfGeoSet named *gset*, call **pfGSetPrimType(***gset***,** *type***)**. Table 10-2 lists the primitive type tokens, the primitive types that they represent, and the number of vertices in a coordinate list for that type of primitive.

Table 10-2 Geometry Primitives

Token	Primitive Type	Number of Vertices
PFGS_POINTS	Points	*numPrims*
PFGS_LINES	Independent line segments	2 * *numPrims*
PFGS_LINESTRIPS	Strips of connected lines	Sum of *lengths* array
PFGS_FLAT_LINESTRIPS	Strips of flat-shaded lines	Sum of *lengths* array
PFGS_TRIS	Independent triangles	3 * *numPrims*

Table 10-2 (continued) Geometry Primitives

Token	Primitive Type	Number of Vertices
PFGS_TRISTRIPS	Strips of connected triangles	Sum of *lengths* array
PFGS_FLAT_TRISTRIPS	Strips of flat-shaded triangles	Sum of *lengths* array
PFGS_QUADS	Independent quadrilaterals	4 * *numPrims*
PFGS_POLYS	Independent polygons	Sum of *lengths* array

where the parameters in the last column represent:

numPrims is the number of primitive items in the pfGeoSet, as set by **pfGSetNumPrims()**.

lengths is the array of strip lengths in the pfGeoSet, as set by **pfGSetPrimLengths()** (note that length is measured here in terms of number of vertices).

Connected primitive types (line strips, triangle strips, and polygons) require a separate array that specifies the number of vertices in each primitive. Length is defined as the number of vertices in a strip for STRIP primitives and is the number of vertices in a polygon for the POLYS primitive type. The number of line segments in a line strip is *numVerts* - 1, while the number of triangles in a triangle strip and polygon is *numVerts* - 2. Use **pfGSetPrimLengths()** to set the length array for strip primitives.

The number of primitives in a pfGeoSet is specified by **pfGSetNumPrims(***gset***,** *num***)**. For strip and polygon primitives, *num* is the number of strips or polygons in *gset*.

pfGeoSet Draw Mode

In addition to the primitive type, **pfGSetDrawMode()** further defines how a primitive is drawn. Triangles, triangle strips, quadrilaterals and polygons can be specified as either filled or as wireframe, where only the outline of the primitive is drawn. Use the PFGS_WIREFRAME argument to enable / disable wireframe mode. Another argument, PFGS_FLATSHADE, specifies how the primitive should be shaded. If flat shading is enabled, each primitive or element in a strip is shaded with a single color.

pfGeoSets are normally processed in *immediate mode* which means that **pfDrawGSet()** sends attributes from the user-supplied attribute arrays to the Graphics Pipeline for rendering. However, this kind of processing is subject to some overhead, particularly if the pfGeoSet contains few primitives. In some cases it may help to use GL *display lists* (this is different from the *libpr* display list type pfDispList) or *compiled mode*. In compiled mode, pfGeoSet attributes are copied from the attribute lists into a special data structure called a display list during a compilation stage. This data structure is highly optimized for efficient transfer to the graphics hardware. However, compiled mode has some major disadvantages:

- compilation is usually costly
- a display list must be recompiled whenever its pfGeoSet's attributes change
- the display list uses extra memory

In general, immediate-mode will offer excellent performance with minimal memory usage and no restrictions on attribute volatility which is a key aspect in may advanced applications.. Despite this, experimentation may show cases where compiled mode offers a performance benefit.

To enable or disable compiled mode, call **pfGSetDrawMode()** with the PFGS_COMPILE_GL token. When enabled, compilation is delayed until the next time the pfGeoSet is drawn with **pfDrawGSet()**. Subsequent calls to **pfDrawGSet()** will then send the compiled pfGeoSet to the graphics hardware.

Primitive Connectivity

A pfGeoSet requires a coordinate array that specifies the world coordinate positions of primitive vertices. This array is either indexed or not, depending on whether a coordinate index list is supplied. If the index list is supplied, it's used to index the coordinate array; if not, the coordinate array is interpreted in a sequential order.

A pfGeoSet's primitive type dictates the connectivity from vertex to vertex to define geometry. Figure 10-1 shows a coordinate array consisting of four coordinates, A, B, C, and D, and the geometry resulting from different primitive types. This example uses index lists that index the coordinate array. Note that the flat-shaded line strip and flat-shaded triangle strip

primitives have the vertices listed in the same order as for the smooth-shaded varieties.

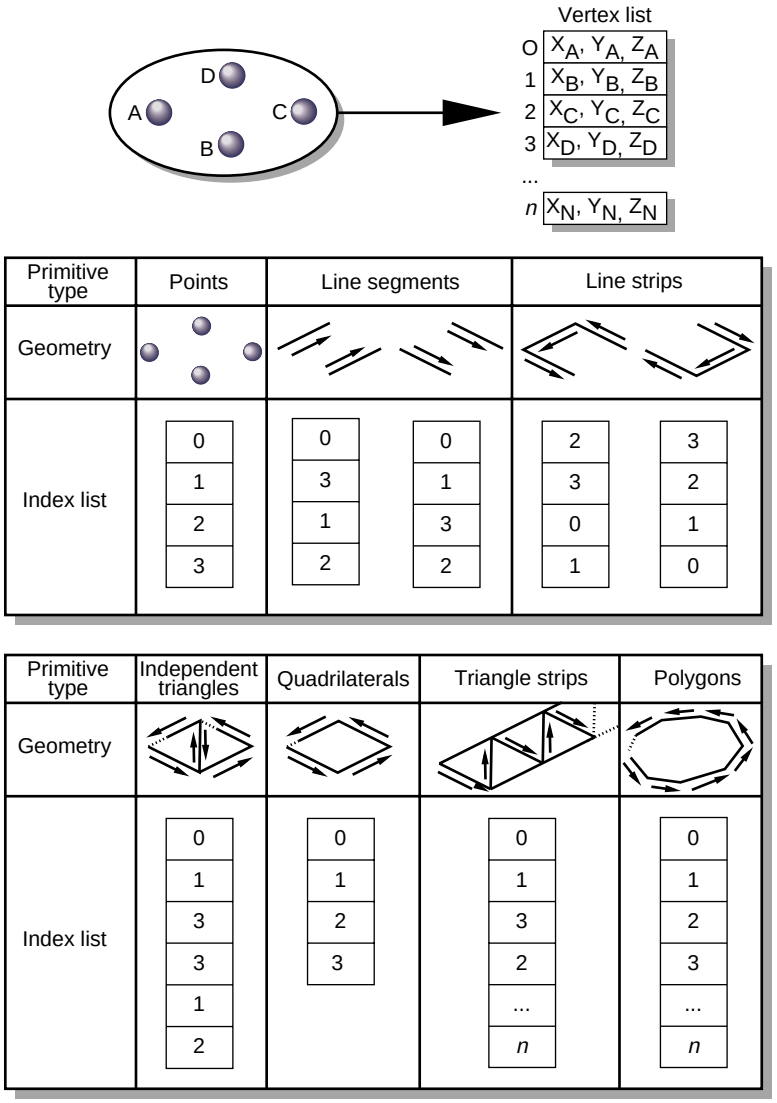


Figure 10-1 Primitives and Connectivity

Attributes

The definition of a primitive isn't complete without attributes. In addition to a primitive type and count, a pfGeoSet references four attribute arrays (see Figure 10-2):

- colors (red, green, blue, alpha)
- normals (Nx, Ny, Nz)
- texture coordinates (S, T)
- vertex coordinates (X, Y, Z)

(A pfGeoState is also associated with each pfGeoSet; see "Graphics State" on page 333 and Figure 10-3 for details.) The four components listed above can be specified with **pfGSetAttr()** and in two ways: by indexed specification—using a pointer to an array of components and a pointer to an array of indices; or by direct specification—providing a NULL pointer for the indices, which indicates that the indices are sequential from the initial value of zero. The choice of indexed or direct components applies to an entire pfGeoSet; that is, all of the supplied components within one pfGeoSet must use the same method. However, you can emulate partially indexed pfGeoSets by using indexed specification and making each nonindexed attribute's index list be a single shared "identity mapping" index array whose elements are 0, 1, 2, 3,..., N-1 where N is the largest number of attributes in any referencing pfGeoSet. (You can share the same array for all such emulated pfGeoSets.). The direct method avoids one level of indirection and may have a performance advantage compared with indexed specification for some combinations of CPU and graphics subsystem.

Note: it is highly recommended that **pfMalloc()** be used to allocate your arrays of attribute data. This will allow IRIS Performer to reference-count the arrays and delete them when appropriate. It will also allow you to easily put your attribute data into shared memory for multiprocessing by specifying an arena such as **pfGetSharedArena()** to **pfMalloc()**. While perhaps convenient, it is very dangerous to specify pointers to static data for pfGeoSet attributes. Early versions of IRIS Performer permitted this but it is strongly discouraged and may have undefined and unfortunate consequences.

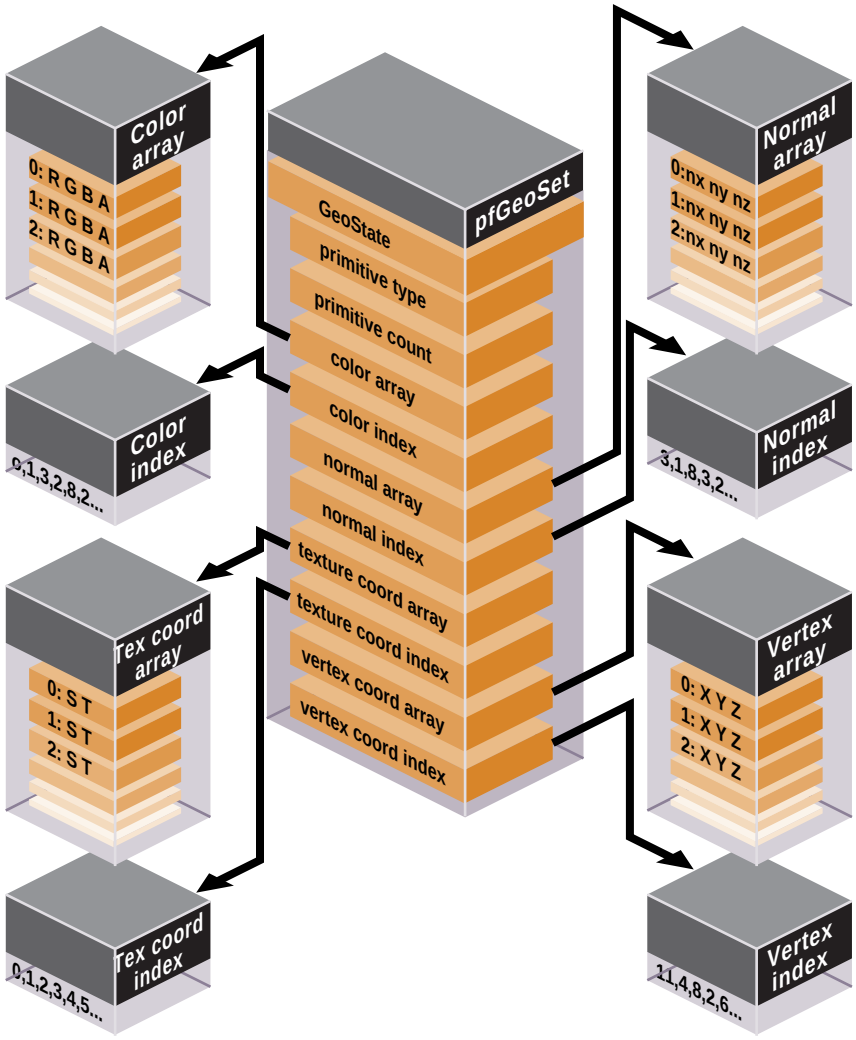


Figure 10-2 pfGeoSet Structure

Attribute Bindings

Attribute bindings specify where in the definition of a primitive an attribute has effect. You can leave a given attribute unspecified; otherwise, its binding location is one of the following:

- overall (one value for the entire pfGeoSet)
- per primitive
- per vertex

Only certain binding types are supported for some attribute types.

Table 10-3 shows the attribute bindings that are legal for each type of attribute.

Table 10-3 Attribute Bindings

Binding Token	Color	Normal	Texture Coordinate	Coordinate
PFGS_OVERALL	Yes	Yes	No	No
PFGS_PER_PRIM	Yes	Yes	No	No
PFGS_PER_VERTEX	Yes	Yes	Yes	Yes
PFGS_OFF	Yes	Yes	Yes	No

Attribute lists, index lists, and binding types are all set by **pfGSetAttr()**.

For FLAT primitives (PFGS_FLAT_TRISTRIPS, PFGS_FLAT_LINESTRIPS), the PFGS_PER_VERTEX binding for normals and colors has slightly different meaning. In these cases, per-vertex colors and normals should not be specified for the first vertex in each line strip or for the first two vertices in each triangle strip since FLAT primitives use the last vertex of each line segment or triangle to compute shading.

pfGeoSet Operations

There are many operations you can perform on pfGeoSets. **pfDrawGSet()** “draws “the indicated pfGeoSet by sending commands and data to the Geometry Pipeline, unless IRIS Performer’s display-list mode is in effect. In

display-list mode, rather than sending the data to the pipeline, the current pfDispList "captures" the **pfDrawGSet()** command. The given pfGeoSet is then drawn along with the rest of the pfDispList with the **pfDrawDList()** command.

When the PFGS_COMPILE_GL mode of a pfGeoSet is not active (**pfGSetDrawMode()**), **pfDrawGSet()** uses rendering loops tuned for each primitive type and attribute binding combination to reduce CPU overhead in transferring the geometry data to the hardware pipeline. Otherwise, **pfDrawGSet()** sends a special, compiled data structure.

Table 10-1 lists other operations that you can perform on pfGeoSets**. pfCopy()** does a shallow copy, copying the source pfGeoSet's attribute arrays by reference and incrementing their reference counts. **pfDelete()** frees the memory of a pfGeoSet and its attribute arrays (if those arrays were allocated with **pfMalloc(**) and provided their reference counts reach zero). **pfPrint()** is strictly a debugging utility and will print a pfGeoSet's contents to a specified destination. **pfGSetIsectSegs()** allows intersection testing of line segments against the geometry in a pfGeoSet; see "Intersecting With pfGeoSets" in Chapter 11 for more information on that function.

3D Text

In addition to the pfGeoSet, *libpr* offers two other primitives which together are useful for rendering a specific type of geometry—three-dimensional characters. See Chapter 5, "Nodes and Node Types" and the description for pfText nodes for an example of how to set up three-dimension text within the context of *libpf*.

pfFont

The basic primitive supporting text rendering is the *libpr* pfFont primitive. A pfFont is essentially a collection of pfGeoSets in which each pfGeoSet represents one character of a particular font. pfFonts also contain metric data, such as a per-character *spacing*, the three-dimensional escapement offset used to increment a text 'cursor' after the character has been drawn. Thus, pfFonts maintain all of the information that is necessary to draw any and all valid characters of a font. However, note that pfFonts are passive and have little functionality on their own, for example you cannot draw a

pfFont—it simply provides the character set for the next higher-level text data object, the pfString.

Table 10-4 lists some routines that are used with a pfFont.

Table 10-4 pfFont Routines

Function	Description
pfNewFont	Create a new pfFont.
pfDelete	Delete a pfFont.
pfFontCharGSet	Set the pfGeoSet to be used for a specific character of this pfFont.
pfFontCharSpacing	Set the 3D spacing to be used to update a text cursor after this character has been rendered.
pfFontMode	Specify a particular mode for this pfFont. Valid Modes to set: **PFFONT_CHAR_SPACING** — specify whether to use fixed or variable spacings for all characters of a pfFont. Possible values are **PFFONT_CHAR_SPACING_FIXED** and **PFFONT_CHAR_SPACING_VARIABLE,** the latter being the default. **PFFONT_NUM_CHARS** — specify how many characters are in this font. **PFFONT_RETURN_CHAR** — specify the index of the character that is considered a ‘return’ character and thus relevant to line justification.
pfFontAttr	Specify a particular attribute of this pfFont. Valid Attributes to set: **PFFONT_NAME** - name of this font. **PFFONT_GSTATE** - pfGeoState to be used when rendering this font. **PFFONT_BBOX** - bounding box that bounds each individual character. **PFFONT_SPACING** - Set the overall character spacing if this is a fixed width font (also the spacing used if one hasn’t been set for a particular character).

Example 10-1 Loading Characters into a pfFont

```
/* Setting up a pfFont */
pfFont *ReadFont(void)
{
    pfFont *fnt = pfNewFont(pfGetSharedArena());
    for(i=0;i<numCharacters;i++)
    {
        pfGeoSet* gset = getCharGSet(i);
        pfVec3* spacing = getCharSpacing(i);

        pfFontCharGSet(fnt, i, gset);
        pfFontCharSpacing(fnt, i, spacing);
    }
}
```

pfString

Simple rendering of three-dimensional text can be done using a pfString. A pfString is an array of font indices stored as 8-bit bytes, 16-bit shorts, or 32-bit integers. Each element of the array contains an index to a particular character of a pfFont structure. A pfString can not be drawn until it has been associated with a pfFont object via a call to **pfStringFont()**. To render a pfString once it references a pfFont, call the function **pfDrawString()**.

pfStrings support the notion of ‘flattening’ to trade off memory for faster processing time. This will cause individual, non-instanced geometry to be used for each character, eliminating the cost of translating the text cursor between each character when drawing the pfString.

Example 10-2 Setting up and drawing a pfString

```
/* Create a string a rotate it for 2.5 seconds */
void
LoadAndDrawString(const char *text)
{
    pfFont *myfont = ReadMyFont();
    pfString *str = pfNewString(NULL);
    pfMatrix mat;
    float start,t;

      /* Use myfont as the 3-d font for this string */
    pfStringFont(str, fnt);
```

```
/* Center String */
pfStringMode(str, PFSTR_JUSTIFY, PFSTR_MIDDLE);

/* Color String is Red */
pfStringColor(str, 1.0f, 0.0f, 0.0f, 1.0f);

/* Set the text of the string */
pfStringString(str, text);

/* Obtain a transform matrix to place this string */
GetTheMatrixToPlaceTheString(mat);
pfStringMat(str, &mat);

/* optimize for draw time by flattening the transforms */
pfFlattenString(str);

/* Twirl text for 2.5 seconds */
start = pfGetTime();
do
{
   pfVec4 clr;
   pfSetVec4(clr, 0.0f, 0.0f, 0.0f, 1.0f);

   /* Clear the screen to black */
   pfClear(PFCL_COLOR|PFCL_DEPTH, clr);

   t = (pfGetTime() - start)/2.5f;
   t = PF_MIN2(t, 1.0f);

   pfMakeRotMat(mat, t * 315.0f, 1.0f, 0.0f, 0.0f);
   pfPostRotMat(mat, mat, t * 720.0f, 0.0f, 1.0f, 0.0f);

   t *= t;
   pfPostTransMat(mat, mat, 0.0f,
          150.0f * t + (1.0f - t) * 800.0f, 0.0f);

   pfPushMatrix();
   pfMultMatrix(mat);

   /* DRAW THE INPUT STRING */
   pfDrawString(str);

   pfPopMatrix();

   pfSwapWinBuffers(pfGetCurWin());
```

```
    } while(t < 2.5f);
}
```

Table 10-5 lists the key routines used to manage pfStrings.

Table 10-5 pfString Routines

Function	Description
pfNewString	Create a new pfString
pfDelete	Delete a pfString.
pfStringFont	Set the pfFont to use when drawing this pfString.
pfStringString	Set the character array that this pfString will represent / render.
pfDrawString	Draw this pfString
pfFlattenString	Flatten all positional translations and the current specification matrix into individual pfGeoSets so that more memory is used, but no matrix transforms or translates have to be done between each character of the pfString.
pfStringColor	Set the color of the pfString.
pfStringMode	Specify a particular mode for this pfString. Valid Modes to set: **PFSTR_JUSTIFY** — set the line justification and has the following possible values: **PFSTR_FIRST** or **PFSTR_LEFT**, **PFSTR_MIDDLE** or **PFSTR_CENTER**, and **PFSTR_LAST** or **PFSTR_RIGHT**. **PFSTR_CHAR_SIZE** — set the number of bytes per character in the input string and has the following possible values: **PFSTR_CHAR**, **PFSTR_SHORT**, **PFSTR_INT**.
pfStringMat	Specify a transform matrix that will affect the entire character string when the pfString is drawn
pfStringSpacing Scale	Specify a scale factor for the escapement translations that happen after each character is drawn. This routine is useful for changing the spacing between characters and even between lines.

Graphics State

The graphics libraries are immediate-mode state machines; if you set a mode, all subsequent geometry is drawn in that mode. For the best performance, mode changes need to be minimized and managed carefully. *libpr* manages a subset of graphics library state and identifies bits of state as *graphics state elements*. Each state element is identified with a PFSTATE token, e.g., PFSTATE_TRANSPARENCY corresponds to the transparency state element. State elements are loosely partitioned into three categories: modes, values and attributes.

Modes are the graphics state variables, such as transparency and texture enable, that have simple values like ON and OFF. An example of a mode command is **pfTransparency(mode)**.

Values are not modal, rather they are real numbers which signify a threshold or quantity. An example of a value is the reference alpha value specified with the **pfAlphaFunc()** command.

Attributes are references to encapsulations (structures) of graphics state. They logically group the more complicated elements of state, such as textures and lighting models. Attributes are structures that are modified through a procedural interface and must be applied to have an effect. For example, **pfApplyTex(tex)** applies the texture map, *tex*, to subsequently drawn geometry.

In *libpr*, there are three methods of setting state:

- immediate mode
- display list mode
- pfGeoState mode

Like the graphics libraries, *libpr* supports the notion of both immediate and display-list modes. In immediate mode, graphics mode changes are sent directly to the Geometry Pipeline, i.e., they have immediate effect. In display-list mode, graphics mode changes are captured by the currently active pfDispList, which can be drawn later. *libpr* display lists differ from graphics library objects because they capture only *libpr* commands and are reusable. *libpr* display lists are useful for multiprocessing applications in

which one process builds up the list of visible geometry and another process draws it. “Display Lists” on page 355 describes *libpr* display lists.

A pfGeoState is a structure that encapsulates all the graphics modes and attributes that *libpr* manages. You can individually set the state elements of a pfGeoState to define a *graphics context*. The act of applying a pfGeoState with **pfApplyGState()** configures the state of the Geometry Pipeline according to the modes, values, and attributes set in the pfGeoState. For example, the following code fragment shows equivalent ways (except for some inheritance properties of pfGeoStates described later) of setting up some lighting parameters suitable for a glass surface:

```
/* Immediate mode state specification */
pfMaterial *shinyMtl;
pfTransparency(PFTR_ON);
pfApplyMtl(shinyMtl);
pfEnable(PFEN_LIGHTING);

/* is equivalent to: */

/* GeoState state specification */
pfGeoState *gstate;
pfGStateMode(gstate, PFSTATE_TRANSPARENCY, PFTR_ON);
pfGStateAttr(gstate, PFSTATE_FRONTMTL, shinyMtl);
pfGStateMode(gstate, PFSTATE_ENLIGHTING, PF_ON);
pfApplyGState(gstate);
```

In addition, pfGeoStates have unique state inheritance capabilities that make them very convenient and efficient; they provide independence from ordered drawing. pfGeoStates are described in the “pfGeoState” section of this chapter.

Libpr routines have been designed to produce an efficient structure for managing graphics state. You can also set graphics state directly through the GL. However, *libpr* will have no record of these settings and will not be able to optimize them and may make incorrect assumptions about current graphics state if the resulting state does not match the *libpr* record when *libpr* routines are called. Therefore, it is best to use the *libpr* routines whenever possible to change graphics state and to restore libpr state if you go directly through the GL.

The following sections will describe the rendering geometry and state elements in detail. There are three types of state elements: modes, values and attributes. Modes are simple settings that take a set of integer values that include values for enabling and disabling the mode. Modes may also have associated values that allow a setting from a defined range. Attributes are complex state structures that encapsulate a related collection of modes and values. Attribute structures will not include in their definition an enable or disable as the enabling or disabling of a mode is orthogonal to the particular related attribute in use.

Rendering Modes

libpr manages a subset of the rendering modes found in the graphics libraries. In addition, *libpr* abstracts certain concepts like transparency, providing a higher-level interface that hides the underlying implementation mechanism.

libpr provides tokens that identify the modes that it manages. These tokens are used by pfGeoStates and other state-related functions like **pfOverride()**. The following table enumerates the PFSTATE_ tokens of supported modes, each with a brief description and default value.

Table 10-6 lists and describes the mode tokens.

Table 10-6 pfGeoState Mode Tokens

Token Name	Description	Default Value
PFSTATE_TRANSPARENCY	Transparency modes	PFTR_OFF
PFSTATE_ALPHAFUNC	Alpha function	PFAF_ALWAYS
PFSTATE_ANTIALIAS	Antialiasing mode	PFAA_OFF
PFSTATE_CULLFACE	Face culling mode	PFCF_OFF
PFSTATE_DECAL	Decaling mode for coplanar geometry	PFDECAL_OFF
PFSTATE_SHADEMODEL	Shading model	PFSM_ GOURAUD
PFSTATE_ENLIGHTING	Lighting enable flag	PF_OFF

Table 10-6 (continued) pfGeoState Mode Tokens

Token Name	Description	Default Value
PFSTATE_ENTEXTURE	Texturing enable flag	PF_OFF
PFSTATE_ENFOG	Fogging enable flag	PF_OFF
PFSTATE_ENWIREFRAME	pfGeoSet wireframe mode enable flag	PF_OFF
PFSTATE_ENCOLORTABLE	pfGeoSet colortable enable flag	PF_OFF
PFSTATE_ENHIGHLIGHTING	pfGeoSet highlighting enable flag	PF_OFF
PFSTATE_ENLPOINTSTATE	pfGeoSet light point state enable flag	PF_OFF
PFSTATE_ENTEXGEN	Texture coordinate generation enable flag	PF_OFF

The mode control functions described in the following sections should be used in place of their graphics library counterparts so that IRIS Performer can correctly track the graphics state. Use **pfGStateMode()** with the appropriate PFSTATE token to set the mode of a pfGeoState.

Transparency

You can control transparency using **pfTransparency()**. Possible transparency modes are:

Table 10-7 pfTransparency Tokens

Transparency mode	Description
PFTR_OFF	Transparency disabled.
PFTR_ON PFTR_FAST	Use the fastest, but not necessarily the best, transparency provided by the hardware.
PFTR_HIGH_QUALITY	Use the best, but not necessarily the fastest, transparency provided by the hardware.

Table 10-7 (continued) pfTransparency Tokens

Transparency mode	Description
PFTR_MS_ALPHA	Use screen-door transparency when multisampling. Fast but limited number of transparency levels.
PFTR_BLEND_ALPHA	Use alpha-based blend with background color. Slower but high number of transparency levels.

In addition, the flag PFTR_NO_OCCLUDE may be logically OR-ed into the transparency mode in which case geometry will not write depth values into the frame buffer. This will prevent it from occluding subsequently rendered geometry. Enabling this flag improves the appearance of unordered, blended transparent surfaces.

There are two basic transparency mechanisms: screen-door transparency which requires hardware multisampling and blending. Blending offers very high quality transparency but for proper results requires that transparent surfaces be rendered in back-to-front order after all opaque geometry has been drawn. When using transparent texture maps to “etch” geometry or if the surface has constant transparency, screen-door transparency is usually good enough. Blended transparency is usually required to avoid “banding” on surfaces with low transparency gradients like clouds and smoke.

Shading Model

Selects flat shading or Gouraud (smooth) shading. **pfShadeModel()** takes one of two tokens: PFSM_FLAT or PFSM_GOURAUD. One some graphics hardware flat shading can offer a significant performance advantage.

Alpha Function

pfAlphaFunc() is an extension of the IRIS GL function **afunction**(3g) and the OpenGL function **glAlphaFunc()**; it allows IRIS Performer to keep track of the hardware mode. The alpha function is a pixel test that compares the incoming alpha to a reference value and uses the result to determine whether or not the pixel is rendered. The reference value must be specified in the range [0, 1]. For example, a pixel whose alpha value is 0 is not rendered if the alpha function is PFAF_GREATER and the alpha reference value is also

0. Note that rejecting pixels based alpha can be faster than using transparency alone. A common technique for improving the performance of filling polygons is to set an alpha function that will reject pixels of low (possibly non-zero) contribution. Alpha function is typically used for see-through textures like trees.

Decals

On Z-buffer based graphics hardware, coplanar geometry can cause unwanted artifacts due to the finite numerical precision of the hardware which cannot accurately resolve which surface has visual priority. This can result in *flimmering*, a visual "tearing" or "twinkling" of the surfaces. **pfDecal()** is used to accurately draw coplanar geometry on IRIS platforms and it supports two implementation methods, each with its advantages:

The *stencil decaling* method uses a hardware resource known as a stencil buffer and requires that a single stencil plane (see IRIS GL **stencil()** and OpenGL **glStencilOp()** man pages) be available for IRIS Performer. This method offers the highest image quality but requires that geometry be coplanar and rendered in a specific order which reduces opportunities for the performance advantage of sorting by graphics mode.

A potentially faster method is the *displace decaling* method. In this case, each layer is displaced towards the eye so it "hovers" slightly above the preceding layer. Displaced decals need not be coplanar, can be drawn in any order but the displacement may cause geometry to incorrectly "poke through" other geometry.

Decals consist of *base geometry* and *layer geometry*. The base defines the depth values of the decal while layer geometry is simply "inlaid" on top of the base. Multiple layers are supported but limited to 8 when using displaced decals. Realize that these layers imply superposition; there is no limit to the number of polygons in a layer, only to the number of distinct layers.

The decal mode indicates whether the subsequent geometry is base or layer and the decal method to use. For example, a mode of PFDECAL_BASE_STENCIL means that subsequent geometry is to be considered as base geometry and drawn using the stencil method. All combinations of base/layer and displace/stencil modes are supported but you should make sure to use the same method for a given base-layer pair.

Example 10-3 illustrates the use of **pfDecal()**.

Example 10-3 Using **pfDecal()** to draw road with stripes

```
pfDecal(PFDECAL_BASE_STENCIL);

/* ... draw underlying geometry (roadway) here ...*/

pfDecal(PFDECAL_LAYER_STENCIL);

/* ... draw coplanar layer geometry (stripes) here ... */

pfDecal(PFDECAL_OFF);
```

Note: *libpf* applications can use the pfLayer node to include decals within a scene graph.

Frontface / Backface

pfCullFace() controls which side of a polygon (if any) is discarded in the Geometry Pipeline. Polygons are either front-facing or back-facing. A front-facing polygon is described by a counterclockwise order of vertices in screen coordinates, and a back-facing one has a clockwise order. **pfCullFace()** has four possible arguments:

PFCF_OFF	Disable face-orientation culling
PFCF_BACK	Cull back-facing polygons
PFCF_FRONT	Cull front-facing polygons
PFCF_BOTH	Cull both front- and back-facing polygons

In particular, backface culling is highly recommended since it offers a significant performance advantage for databases where polygons are never be seen from both sides (databases of “solid” objects or with constrained eyepoints).

Antialiasing

pfAntialias() is used to turn the antialiasing mode of the hardware on or off. Currently, antialiasing is implemented differently by each different graphics system. Antialiasing can produce artifacts as a result of the way IRIS Performer and the active hardware platform implement the feature. See the reference page for **pfAntialias()** for implementation details.

Rendering Values

Some modes may also have associated values. These values are set through **pfGStateVal().** Table 10-8 lists and describes the value tokens.

Table 10-8 pfGeoState Value Tokens

Token Name	Description	Range	Default Value
PFSTATE_ALPHAREF	Set the alpha function reference value.	0.0 - 1.0	0.0

Enable / Disable

pfEnable() and **pfDisable()** control certain rendering modes. Certain modes do not have effect when enabled but require that other attribute(s) be applied. Table 10-9 lists and describes the tokens and also lists the attributes required for the mode to become truly active.

Table 10-9 Enable and Disable Tokens

Token	Action	Attribute(s) Required
PFEN_LIGHTING	Enable or disable lighting.	pfMaterial pfLight pfLightModel
PFEN_TEXTURE	Enable or disable texture.	pfTexEnv pfTexture
PFEN_FOG	Enable or disable fog.	pfFog

Table 10-9 (continued) Enable and Disable Tokens

Token	Action	Attribute(s) Required
PFEN_WIREFRAME	Enable or disable pfGeoSet wireframe rendering.	none
PFEN_COLORTABLE	Enable or disable pfGeoSet colortable mode.	pfColortable
PFEN_HIGHLIGHTING	Enable or disable pfGeoSet highlighting.	pfHighlight
PFEN_TEXGEN	Enable or disable automatic texture coordinate generation.	pfTexGen
PFEN_LPOINTSTATE	Enable or disable pfGeoSet light points	pfLPointState

By default all modes are disabled.

Rendering Attributes

Rendering attributes are state structures that are manipulated through a procedural interface. Examples include pfTexture, pfMaterial, and pfFog. *libpr* provides tokens that enumerate the graphics attributes it manages. These tokens are used by pfGeoStates and other state-related functions like **pfOverride()**. Table 10-10 lists and describes the tokens.

Table 10-10 Rendering Attribute Tokens

Attribute Token	Description	Apply Routine
PFSTATE_LIGHTMODEL	Lighting model	pfApplyLModel
PFSTATE_LIGHTS	Light source definitions	pfLightOn
PFSTATE_FRONTMTL	Front-face material	pfApplyMtl
PFSTATE_BACKMTL	Back-face material	pfApplyMtl
PFSTATE_TEXTURE	Texture	pfApplyTex

Table 10-10 (continued) Rendering Attribute Tokens

Attribute Token	Description	Apply Routine
PFSTATE_TEXENV	Texture environment	pfApplyTEnv
PFSTATE_FOG	Fog model	pfApplyFog
PFSTATE_COLORTABLE	Color table for pfGeoSets	pfApplyCtab
PFSTATE_HIGHLIGHT	Definition of pfGeoSet highlighting style	pfApplyHlight
PFSTATE_LPOINTSTATE	pfGeoSet light point definition	pfApplyLPState
PFSTATE_TEXGEN	Texture coordinate generation definition	pfApplyTGen

Rendering attributes control which attributes are applied to geometric primitives when they're processed by the hardware. All IRIS Performer attributes consist of a control structure, definition routines, and an apply function, **pfApply*** (except for lights which are "turned on").

Each attribute has an associated **pfNew*()** routine that allocates storage for the control structure. When sharing attributes across processors in a multiprocessor application, you should pass the **pfNew*()** routine a shared memory arena from which to allocate the structure. If you pass NULL as the arena, the attribute is allocated from the heap and isn't sharable in a non-shared address space (**fork()**) multiprocessing application.

All attributes can be applied directly, referenced by a pfGeoState or captured by a display list. When changing an attribute, that change isn't visible until the attribute is reapplied. Detailed coverage of attribute implementation is available in the reference pages.

Texture

IRIS Performer supports texturing through pfTextures and pfTexEnvs, which are roughly analogous to graphics library textures (see **texdef2d**() for IRIS GL, or **glTexImage2D()** for OpenGL) and texture environments (see IRIS GL's **tevdef**(3g) or OpenGL's **glTexEnv()**). A pfTexture defines a texture image, format, and filtering. A pfTexEnv specifies how the texture should interact with the colors of the geometry it's applied to. You need both to

display textured data, but you don't need to specify them both at the same time. For example, you could have pfGeoStates each of which had a different texture specified as an attribute and still use an overall texture environment specified with **pfApplyTEnv()**.

A pfTexture is created by calling **pfNewTex()**. If the desired texture image exists as a disk file in IRIS libimage format (the file often has a ".rgb" suffix), you can call **pfLoadTexFile()** to load the image into CPU memory and set the image format. Otherwise, **pfTexImage()** lets you directly provide an image array in the same external format as specified on the pfTexture and as expected by **texdef2d()** in IRIS GL and **glTexImage2D()** in OpenGL. IRIS GL and OpenGL both expect packed texture data with each row beginning on a long word boundary. However, IRIS GL and OpenGL expect the individual components of a texel to be packed in opposite order. For example, IRIS GL expects four component texels to be packed as ABGR OpenGL expects the texels to be packed as RGBA. If you provide your own image array in a multiprocessing environment, it should be allocated from shared memory (along with your pfTexture) to allow different processes to access it.

Note: The size of your texture must be an integral power of two on each side. IRIS GL scales up your textures to the next power of two, making them take up to four times more space in hardware texture memory! OpenGL simply refuses to accept badly sized textures. You can rescale your texture images with the *izoom* or *imgworks* programs (shipped with IRIX 5.3 in the eoe2.sw.imagetools and imgtools.sw.tools subsystems; and with IRIX 6.2 in the eoe.sw.imagetools and imgworks.sw.tools subsystems).

Your texture source does not have to be a static image. **pfTexLoadMode()** can be used to set one of the sources listed in Table 10-11 with PFTEX_LOAD_BASE. Note that sources other than CPU memory may not be supported on all graphics platforms, or may have some special restrictions. The sample program, */usr/share/Performer/src/pguide/libpr/C/movietex.c* demonstrates the use of different texture sources for playing texture or video movies on pfGeoSet

"screens". Changing your image source may cause your texture to go through the formatting stage upon the next **pfApplyTex()**.

Table 10-11 Texture Image Sources

PFTEX_SOURCE_ Token	Texture image is take from:
IMAGE	CPU memory location specified by **pfTexLoadImage()** or **pfTexImage()**.
FRAMEBUFFER	framebuffer location offset from window origin as specified by **pfTexLoadOrigin()**.
VIDEO	Sirius video texture drain.

Texture storage is limited only by virtual memory, but for real-time applications you must consider the amount of texture storage the graphics hardware supports. Textures that don't fit in the graphics subsystem will be paged as needed when **pfApplyTex()** is called. *Libpr* provides routines for managing hardware texture memory so that a real-time application does not have to get a surprise texture load. **pfIsTexLoaded()**, called from the drawing process, will tell you if the pfTexture is currently properly loaded in texture memory. **pfIdleTex()** can be used to free up the hardware texture memory owned by a pfTexture.

pfLoadTex(), called from the drawing process, can be used to explicitly load a texture into graphics hardware texture memory (which will include doing any necessary formatting of the texture image). By default, **pfLoadTex()** will load the entire texture image, including any required minification or magnification levels, into texture memory. **pfSubloadTex()** and **pfSubloadTexLevel()** can also be used in the drawing process to do an immediate load of texture memory managed by the given pfTexture and these routines allow you to specify all loading parameters (source, origin, size, etc.). This is useful for loading different images for the same pfTexture in different graphics pipelines.

A special **pfTexFormat()** formatting mode, PFTEX_SUBLOAD_FORMAT, allows part or all of the image in texture memory owned by the pfTexture to be replaced via **pfApplyTex()**, **pfLoadTex()**, or **pfSubloadTex()**, without having to go through the expensive reformatting phase. This allows you to quickly update the image of a pfTexture in texture memory. The PFTEX_SUBLOAD_FORMAT used with an appropriate **pfTexLoadSize()**

and **pfTexLoadOrigin()** allows you to control what part of the texture will be loaded by subsequent calls to **pfLoadTex()** or **pfApplyTex()**. There are also different loading modes that cause **pfApplyTex()** to automatically reload or subload a texture from a specified source. If you want the image of a pfTexture to be updated upon every call to **pfApplyTex()**, you can set the loading mode of the pfTexture with **pfTexLoadMode()** to be PFTEX_BASE_AUTO_REPLACE. **pfTexLoadImage()** allows you to continuously update the memory location of an IMAGE source texture without triggering any reformatting of the texture.

Note: In IRIS GL, the SUBLOAD format (same as the FAST_DEFINE format on old versions of IRIS Performer) can only be used on non-MIPmapped textures. The fast texture loading uses the IRIS GL **subtexload()** and OpenGL **glTexSubImage()** calls. In IRIS GL, subload sizes must be integral multiples of 32.

Hint: There are additional texture formatting modes that can improve texture performance. Of most importance is the 16-bit texel internal formats. These formats cause the resulting texels to have 16 bits of resolution instead of the standard 32. These formats can have dramatically faster texture fill performance and cause the texture to take up half the hardware texture memory. Therefore, they are strongly recommended and are used by default. There are different formats for each possible number of components to give a choice of how the compression is to be done. These formats are described in the **pfTexFormat**(3pf) reference page.

There may also be formatting modes for internal or external image formats that IRIS Performer does not have a token for. However, the GL value can be specified. Specifying GL values will make your application GL specific and may also cause future porting problems, so it should only be done if absolutely necessary.

pfTextures also allow you to define a set of textures that are mutually exclusive, should always be applied to the same set of geometry, and thus that can share the same location in hardware texture memory. With **pfTexList(***tex, list***)** you can specify a list of textures to be in a texture set managed by the base texture, *tex*. The base texture is what gets applied with **pfApplyTex()**, or assigned to geometry through pfGeoStates. With **pfTexFrame()**, you can select a given texture from the list (-1 selects the base texture and is the default). This allows you to define a texture movie where each image is the frame of the movie. You can have an image on the base

texture to display when the movie is not playing. There are additional loading modes for **pfTexLoadMode()** described in Table 10-12 to control how the textures in the texture list share memory with the base texture.

Table 10-12 Texture Load Modes

PFTEX_LOAD_ modeToken	Load mode values	Description:
BASE	BASE_APPLY BASE_AUTO_SUBLOAD	Loading of image is done as required for pfApply, or automatically subloaded upon every pfApply.
LIST	LIST_APPLY LIST_AUTO_IDLE LIST_AUTO_SUBLOAD	Loading of list texture image is as separate apply, causes freeing of previous list texture in hardware texture memory, or is subloaded into memory managed by the base texture.

Texture list textures can share the exact graphics texture memory as the base texture but this has the restriction that the textures must all be the exact same size and format as the base texture. Texture list textures can also indicate that they are mutually exclusive which will cause the texture memory of previous textures to be freed before applying the new texture. This method has no restrictions on the texture list, but is less efficient than the previous method. Finally, texture list textures can be treated as completely independent textures that should all be kept resident in memory for rapid access upon their application.

pfTexFilter() sets a desired filter on a pfTexture. The minification and magnification texture filters are described with bitmask tokens. If filters are partially specified, IRIS Performer will fill in the rest with machine dependent fast defaults. The PFTEX_FAST token can be included in the bitmask to allow IRIS Performer to make machine dependent substitutions where there are large performance differences.

There are a variety of texture filter functions that can improve the look of textures when they are minified and magnified. By default, textures use MIPmapping when minified (though this costs an extra 1/3 in storage space to store the minification levels). Each level of minification or magnification of a texture is twice the size of the previous level. Minification levels are

indicated with positive numbers and magnification levels are indicated with non-positive numbers. The default magnification filter for textures is bilinear interpolation. The use of detail textures and sharpening filters can improve the look of magnified textures. Detailing actually uses an extra detail texture that you provide that is based on a specified level of magnification from the corresponding base texture. The detail texture can be specified with the **pfTexDetail()** command. By default, MIPmap levels are generated for the texture automatically. OpenGL operation allows for the specification of custom MIPmap levels. Both MIPmap levels and detail levels can be specified with **pfTexLevel()**. The level number should be a positive number for a minification level and a non-positive number for a magnification (detail) level. If you are providing your own minification levels, you must provide all $\log_2(\text{MAX}(texSizeX, texSizeY))$ minification levels. There is only one detail texture for a pfTexture.

The magnification filters use spline functions to control their rate of application as a function of magnification, and specified level of magnification for detail textures. These splines can be specified with **pfTexSpline()**. The specification of the spline is a set of control points that are pairs of non-decreasing magnification levels (specified with non-positive numbers) and corresponding scaling factors. Magnification filters can be applied to all components of a texture, only the RGB components of a texture, or to just the alpha components. OpenGL does not allow different magnification filters (between detail and sharpen) for RGB and alpha channels.

Note: The specification of detail textures may have GL dependencies and magnifications filters may not be available on all hardware configurations. The pfTexture reference page describes these details.

Automatic Texture Coordinate Generation

Automatic texture coordinate generation is provided with the pfTexGen state attribute. pfTexGen closely corresponds to IRIS GL's **texgen()** and OpenGL's **glTexGen()** functions. When texture coordinate generation is enabled, a pfTexGen applied with **pfApplyTGen()** will automatically generate texture coordinates for all rendered geometry. Texture coordinates are generated from geometry vertices according to the texture generation mode set with **pfTGenMode()**. Available modes and their function are listed

in Table 10-13. Some modes refer to a plane which is set with **pfTGenPlane()**.

Table 10-13 Texture generation modes

PFTG_ Mode Token	Texture coordinates are calculated as...
OBJECT_PLANE	distance of vertex from plane in object coordinates
EYE_PLANE	distance of vertex from plane in eye coordinates. The plane is transformed by the inverse of the ModelView matrix when the pfTexGen is applied.
EYE_PLANE_IDENT	distance of vertex from plane in eye coordinates. The plane is not transformed by the inverse of the ModelView matrix when the pfTexGen is applied
SPHERE_MAP	an index into a 2D reflection map based on vertex position and normal. Specifics of the calculation are found in the graphics libraries' man pages.

Lighting

IRIS Performer lighting is an extension of graphics library lighting (see **lmdef**(3g) for IRIS GL or **glLight()** and related functions in OpenGL), but IRIS Performer divides the actions of **lmdef()** into lights and light models, just as OpenGL does. The light embodies the color, position, and type (for example, infinite or spot) of the light. The light model specifies the environment for infinite (the default) or local viewing, and two-sided illumination. pfLights and pfLightModels are created by calling **pfNewLight()** and **pfNewLModel()**, respectively.

The transformation matrix that is on the matrix stack at the time the light is applied controls the interpretation of the light source direction:

1. To attach a light to the viewer (like a miner's head-mounted light), call **pfLightOn()** only once with an identity matrix on the stack.
2. To attach a light to the world (like the sun or moon), call **pfLightOn()** every frame with only the viewing transformation on the stack.
3. To attach a light to an object (like the headlights of a car), call **pfLightOn()** every frame with the combined viewing and modeling transformation on the stack.

The number of lights you can have turned on at any one time is limited by PF_MAX_LIGHTS, just as is true with the graphics libraries.

Note: In IRIS GL, attenuation is also part of the light model definition. In OpenGL, attenuation is defined per-light. There is separate *libpr* API for setting each of these: **pfLModelAtten()** for IRIS GL and **pfLightAtten()** for OpenGL. You can use **pfQueryFeature()** with a feature specifier value of PFQFTR_LMODEL_ATTENUATION or PFQFTR_LIGHT_ATTENUATION to find out which is supported in the current run-time environment.

Note: *libpf* applications can include light sources in a scene graph with the pfLightSource node.

Materials

IRIS Performer materials are an extension of graphics library materials (see **lmdef**(3g) for IRIS GL or **glMaterial()** for OpenGL). pfMaterials encapsulate the ambient, diffuse, specular, and emissive colors of an object as well as its *shininess* and transparency. A pfMaterial is created by calling **pfNewMtl()**. As with any of the other attributes, a pfMaterial can be referenced in a pfGeoState, captured by a display list, or invoked as an immediate mode command.

pfMaterials, by default, allow object colors to set the ambient and diffuse colors. This allows the same pfMaterial to be used for objects of different colors, removing the need for material changes and thus improving performance. This mode can be changed with **pfMtlColorMode(mtl, side, PFMTL_CMODE_*)**. IRIS GL only supports the front material tracking the current color while OpenGL allows front or back materials to track the current color. If the same material is used for both front and back materials, there is no difference in functionality.

Color Tables

A pfColortable substitutes its own color array for the normal color attribute array (PFGS_COLOR4) of a pfGeoSet. This allows the same geometry to appear differently in different views simply by applying a different pfColortable for each view. By leaving the selection of color tables to the global state, you can use a single call to switch color tables for an entire

scene. In this way, color tables can simulate time-of-day changes, infrared imaging, psychedelia, and other effects.

pfNewCtab() creates and returns a handle to a pfColortable. As with other attributes, you can specify which color table to use in a pfGeoState or you can use **pfApplyCtab()** to set the global color table, either in immediate mode or in a display list. For an applied colortable to have effect, colortable mode must also be enabled.

Fog

A pfFog is created by calling **pfNewFog()**. As with any of the other attributes, a pfFog can be referenced in a pfGeoState, captured by a display list, or invoked as an immediate mode command. Fog is the atmospheric effect of aerosol water particles that occlude vision over distance. The IRIS hardware can simulate this phenomenon in several different fashions. A fog color is blended with the resultant pixel color based on the range from the viewpoint and the fog function. pfFog supports several different fogging methods. Table 10-14 lists the pfFog tokens and their corresponding actions.

Table 10-14 pfFog Tokens

pfFog Token	Action
PFFOG_VTX_LIN	Compute fog linearly at vertices.
PFFOG_VTX_EXP	Compute fog exponentially at vertices (e^x).
PFFOG_VTX_EXP2	Compute fog exponentially at vertices ($e^{x \text{ squared}}$).
PFFOG_PIX_LIN	Compute fog linearly at pixels.
PFFOG_PIX_EXP	Compute fog exponentially at pixels (e^x).
PFFOG_PIX_EXP2	Compute fog exponentially at pixels ($e^{x \text{ squared}}$).
PFFOG_PIX_SPLINE	Compute fog using a spline function at pixels.

pfFogType() uses these tokens to set the type of fog. A detailed explanation of fog types is given in the reference page for **pfFog**(3pf) and the IRIS GL **fogvertex**(3g) and OpenGL **glFog**(3g) reference pages.

You can set the near and far edges of the fog with **pfFogRange()**. For exponential fog functions, the near edge of fog is always zero in eye coordinates. The near edge is where the onset of fog blending occurs, and the far edge is where all pixels are 100% fog color.

The token PFFOG_PIX_SPLINE selects a spline function to be applied when generating the hardware fog tables. This is further described in the **pfFog**(3pf) reference page. Spline fog allows the user to define an arbitrary fog ramp that can more closely simulate real-world phenomena like horizon haze.

For best fogging effects the ratio of the far to the near clipping planes should be minimized. In general, it's more effective to add a small amount to the near plane than to reduce the far plane.

Highlights

IRIS Performer provides a mechanism for highlighting geometry with alternative rendering styles, useful for debugging and interactivity. A pfHighlight, created with **pfNewHlight()**, encapsulates the state elements and modes for these rendering styles. A pfHighlight can be applied to an individual pfGeoSet with **pfGSetHlight()**, or can be applied to multiple pfGeoStates through a pfGeoState or **pfApplyHlight()**. The highlighting effects are added to the normal rendering phase of the geometry. pfHighlights make use of special outlining and fill modes and have a concept of a foreground color and a background color that can both be set with **pfHlightColor()**. The available rendering styles can be combined by OR-ing together tokens for **pfHlightMode()** and are described in Table 10-15.

Table 10-15 **pfHlightMode()** Tokens

PFHL_ Mode Bitmask Token	Description
LINES	Outlines the triangles in the highlight foreground color according to **pfHlightLineWidth()**.
LINESPAT LINESPAT2	Outlines triangles with patterned lines in the highlight foreground color, or in two colors using the background color.

Table 10-15 (continued) **pfHlightMode()** Tokens

PFHL_ Mode Bitmask Token	Description
FILL	Draws geometry with the highlight foreground color. Combined with SKIP_BASE, this is a fast highlighting mode.
FILLPAT FILLPAT2	Draws the highlighted geometry as patterned with one or two colors.
FILLTEX	Draw highlighting fill pass with a special highlight texture.
LINES_R FILL_R	Reverses the highlighting foreground and background colors for lines and fill, respectively.
POINTS	Renders the vertices of the geometry as points according to **pfHlightPntSize()**.
NORMALS	Displays the normals of the geometry with lines according to **pfHlightNormalLength()**.
BBOX_LINES BBOX_FILL	Displays the bounding box of the pfGeoSet as outlines and / or filled box. Combined with PFHL_SKIP_BASE, this is a fast highlighting mode.
SKIP_BASE	Causes the normal drawing phase of the pfGeoSet to be skipped. This is recommended when using PFHL_FILL or PFHL_BBOX_FILL.

For a demonstration of the highlighting styles, see the sample program, */usr/share/Performer/pguide/src/libpr/C/hlcube.c*.

Graphics Library Matrix Routines

IRIS Performer provides extensions to the standard graphics library matrix-manipulation functions. These functions are similar to their graphics library counterparts, with the exception that they can be placed in IRIS

Performer display lists. Table 10-16 lists and describes the matrix manipulation routines.

Table 10-16 Matrix Manipulation Routines

Routines	Action
pfScale	Concatenate a scaling matrix.
pfTranslate	Concatenate a translation matrix.
pfRotate	Concatenate a rotation matrix.
pfPushMatrix	Push down the matrix stack.
pfPushIdentMatrix	Push the matrix stack and load an identity matrix on top.
pfPopMatrix	Pop the matrix stack.
pfLoadMatrix	Add a matrix to the top of the stack.
pfMultMatrix	Concatenate a matrix.

Sprite Transformations

A sprite is a special transformation used to efficiently render complex geometry with axial or point symmetry. A classic sprite example is a tree which is rendered as a single, texture-mapped quadrilateral. The texture image is of a tree and has an alpha component whose values which “etches” the tree shape into the quad. In this case, the sprite transformation rotates the quad around the tree trunk axis so that it always faces the viewer. Another example is a puff of smoke which again is a texture-mapped quad but is rotated about a point to face the viewer so it appears the same from any viewing angle. The pfSprite transformation mechanism supports both these simple examples as well as more complicated ones involving arbitrary 3D geometry.

A pfSprite is a structure which is manipulated through a procedural interface. It is different from “attributes” like pfTexture and pfMaterial since it affects transformation, rather than state related to appearance. A pfSprite is activated with **pfBeginSprite()**. This enables “sprite mode” and any pfGeoSet that is drawn before sprite mode is ended with **pfEndSprite()** will be transformed by the pfSprite. First, the pfGeoSet is translated to the

location specified with **pfPositionSprite()**. Then, it is rotated, either about the sprite position or axis depending on the pfSprite's configuration. Note that **pfBeginSprite()**, **pfPositionSprite()** and **pfEndSprite()** are display listable and this will be captured by any active pfDispList.

A pfSprite's rotation mode is set by specifying the PFSPRITE_ROT token to **pfSpriteMode()**. In all modes, the Y axis of the geometry is rotated to point to the eye position. Rotation modes are listed below.

Table 10-17 pfSprite rotation modes

PFSPRITE_ Rotation Token	Rotation Characteristics
AXIAL_ROT	Geometry's Z axis is rotated about the axis specified with **pfSpriteAxis()**.
POINT_ROT_EYE	Geometry is rotated about the sprite position with the object coordinate Z axis constrained to the window coordinate Y axis, i.e., geometry's Z axis stays "upright".
POINT_ROT_WORLD	Geometry is rotated about the sprite position with the object coordinate Z axis constrained to the sprite axis.

Rather than using the graphics hardware's matrix stack, pfSprites transform small pfGeoSets on the CPU for improved performance. However, when a pfGeoSet contains a certain number of primitives it becomes more efficient to use the hardware matrix stack. While this threshold is dependent on the CPU and graphics hardware used, you may specify it with the PFSPRITE_MATRIX_THRESHOLD token to **pfSpriteMode()**. The corresponding value is the minimum vertex requirement for hardware matrix transformation. Any pfGeoSet with fewer vertices will be transformed on the CPU. If you want a pfSprite to affect non-pfGeoSet geometry you should set the matrix threshold to zero so that the pfSprite will always use the matrix stack. When using the matrix stack, **pfBeginSprite()** pushes the stack and **pfEndSprite()** pops the matrix stack so the sprite transformation is limited in scope.

pfSprites are dependent on the viewing location and orientation and the current modeling transformation. You can specify these with calls to **pfViewMat()** and **pfModelMat()** respectively. Note that *libpf*-based applications need not call these routines since *libpf* does it automatically.

Display Lists

libpr supports display lists, which can capture and later execute *libpr* graphics commands. **pfNewDList()** creates and returns a handle to a new pfDispList. A pfDispList can be selected as the current display list with **pfOpenDList()**, which puts the system in display list mode. Any subsequent *libpr* graphics commands, such as **pfTransparency()**, **pfApplyTex()**, or **pfDrawGSet()**, are added to the current display list. Commands are added until **pfCloseDList()** returns the system to immediate mode. It is not legal to have multiple pfDispLists open at a given time but a pfDispList may be reopened in which case commands are appended to the end of the list.

Once a display list is constructed, it can be executed by calling **pfDrawDList()**, which traverses the list and sends commands down the Geometry Pipeline.

pfDispLists are designed for multiprocessing, where one process builds a display list of the visible scene and another process draws it. The function **pfResetDList()** facilitates this by making pfDispLists reusable. Commands added to a reset display list overwrite any previously entered commands. A display list is typically reset at the beginning of a frame and then filled with the visible scene.

pfDispLists support concurrent multiprocessing, where the producer and consumer processes simultaneously write and read the display list. The PFDL_RING argument to **pfNewDList()** creates a ring buffer or FIFO-type display list. pfDispLists automatically ensure ring buffer consistency by providing synchronization and mutual exclusion to processes on ring buffer full or empty conditions.

State Management

pfState is a structure that represents the entire *libpr* graphics state. A pfState maintains a stack of graphics states that can be pushed and popped to save and restore state. The top of the stack describes the current graphics state of a window as it's known to IRIS Performer.

pfInitState() initializes internal *libpr* state structures and should be called at the beginning of an application before any pfStates are created. Multiprocessing applications should pass a **usinit()** semaphore arena

pointer to **pfInitState()**, such as **pfGetSemaArena()**, so IRIS Performer can safely manage state between processes. **pfNewState()** creates and returns a handle to a new pfState, which is typically used to define the state of a single window. If using pfWindows, discussed in “Windows” on page 363, a pfState is automatically created for the pfWindow when the window is opened and the current pfState is switched when the current pfWindow changes. **pfSelectState()** can be used to efficiently switch a different complete pfState. **pfLoadState()** will force the full application of a pfState.

Pushing and Popping State

pfPushState() pushes the state stack of the currently active pfState, duplicating the top state. Subsequent modifications of the state through *libpr* routines are recorded in the top of stack. Consequently, a call to **pfPopState()** restores the state elements that were modified after **pfPushState()**.

The code fragment in Example 10-4 illustrates how to push and pop state.

Example 10-4 Pushing and Popping Graphics State

```
/* set state to transparency=off and texture=brickTex */
pfTransparency(PFTR_OFF);
pfApplyTex(brickTex);

/* ... draw geometry here using original state ... */

/* save old state. establish new state */
pfPushState();
pfTransparency(PFTR_ON);
pfApplyTex(woodTex);

/* ... draw geometry here using new state ...*/

/* restore state to transparency=off and texture=brickTex */
pfPopState();
```

State Override

pfOverride() implements a global override feature for *libpr* graphics state and attributes. **pfOverride()** takes a mask that indicates which state elements to affect and a value specifying whether the elements should be overridden. The mask is a bitwise OR of the state tokens listed previously.

The values of the state elements at the time of overriding become fixed and cannot be changed until **pfOverride()** is called again with a value of zero to release the state elements.

The code fragment in Example 10-5 illustrates the use of **pfOverride()**.

Example 10-5 Using **pfOverride()**

```
pfTransparency(PFTR_OFF);
pfApplyTex(brickTex);

/*
 * Transparency will be disabled and only the brick texture
 * will be applied to subsequent geometry.
 */
pfOverride(PFSTATE_TRANSPARENCY | PFSTATE_TEXTURE, 1);
/* Draw geometry */

/* Transparency and texture can now be changed */
pfOverride(PFSTATE_TRANSPARENCY | PFSTATE_TEXTURE, 0);
```

pfGeoState

A pfGeoState encapsulates all the rendering modes, values, and attributes managed by *libpr*. See “Rendering Modes” on page 335, “Rendering Values” on page 340, and “Rendering Attributes” on page 341 for more information. pfGeoStates provide a mechanism for combining state into logical units and define the appearance of geometry. For example, you can set a brick-like texture and a reddish-orange material on a pfGeoSet and use it when drawing brick buildings.

Local and Global State

There are two levels of rendering state: local and global. A record of both is kept in the current pfState. The local state is that defined by the settings of the current pfGeoState. The rendering state and attributes of a pfGeoState can be either locally set or globally inherited. If all state elements are set locally, a pfGeoState becomes a full graphics context—that is, all state is then defined at the pfGeoState level. Global state elements are set with libpr immediate mode routines like **pfEnable(), pfApplyTex()**, **pfDecal()**, **pfTransparency()** or by drawing a pfDispList containing these commands with **pfDrawDList()**. Local state elements are set by applying a pfGeoState with **pfApplyGState()** (note that **pfDrawGSet()** automatically calls **pfApplyGState()** if the pfGeoSet has an attached pfGeoState). The state elements applied by a pfGeoState are those modes, enables, and attributes that are explicitly set on the pfGeoState.

Note: By default, all state elements are inherited from the global state. Inherited state elements are evaluated faster than values that have been explicitly set.

While it can be useful to have all state defined at the pfGeoState level, it usually makes sense to inherit most state from global default values and then explicitly set only those state elements that are expected to change often.

Examples of useful global defaults are lighting model, lights, texture environment, and fog. Highly variable state is likely to be limited to a small set such as textures, materials, and transparency. For example, if the majority of your database is lighted, simply configure and enable lighting at the beginning of your application. All pfGeoStates will be lighted except the ones for which you explicitly disable lighting. Then attach different pfMaterials and pfTextures to pfGeoStates to define specific state combinations.

Note: Caution should be used when enabling modes in the global state. These modes may have cost even when they have no visible effect. Therefore, geometry that cannot use these modes should have a pfGeoState that explicitly disables the mode. Modes to be especially careful of include the texturing enable and transparency.

You specify that a pfGeoState should inherit state elements from the global default with **pfGStateInherit(gstate, mask)**. *mask* is a bitmask of tokens that indicates which state elements to inherit. These tokens are listed in the "Rendering Modes", "Rendering Values", and "Rendering Attributes" sections of this chapter. For example, PFSTATE_ENLIGHTING | PFSTATE_ENTEXTURE makes *gstate* inherit the enable modes for lighting and texturing.

A state element ceases to be inherited when it is set in a pfGeoState. Rendering modes, values, and attributes are set with **pfGStateMode()**, **pfGStateVal(),** and **pfGStateAttr()**, respectively. For example, to specify that *gstate* is transparent and textured with *treeTex*, use

```
pfGStateMode(gstate, PFSTATE_TRANSPARENCY, PFTR_ON);
pfGStateAttr(gstate, PFSTATE_TEXTURE, treeTex);
```

Applying pfGeoStates

Use **pfApplyGState()** to apply the state encapsulated by a pfGeoState to the Geometry Pipeline. The effect of applying a pfGeoState is similar to applying each state element individually. For example, if you set a pfTexture and enable a decal mode on a pfGeoState, applying it essentially calls **pfApplyTex()** and **pfDecal()**. If in display-list mode, **pfApplyGState()** is captured by the current display list.

State is (logically) pushed before, and popped after, pfGeoStates are applied, so that pfGeoStates don't inherit state from each other. This is a very powerful and convenient characteristic since as a result, pfGeoStates are order-independent, and you don't have to worry about one pfGeoState corrupting another. The code fragment in Example 10-6 illustrates how pfGeoStates inherit state.

Example 10-6 Inheriting State

```
/* gstateA should be textured */
pfGStateMode(gstateA, PFSTATE_ENTEXTURE, PF_ON);

/* gstateB inherits the global texture enable mode */
pfGStateInherit(gstateB, PFSTATE_ENTEXTURE);

/* Texturing is disabled as the global default */
pfDisable(PFEN_TEXTURE);
```

```
/* Texturing is enabled when gstateA is applied */
pfApplyGState(gstateA);
/* Draw geometry that will be textured */

/* The global texture enable mode of OFF is restored
so that gstateB is NOT textured. */
pfApplyGState(gstateB);
/* Draw geometry that will not be textured */
```

The actual pfGeoState pop is a lazy pop that doesn't happen unless a subsequent pfGeoState requires the global state to be restored. This means that the actual state between pfGeoStates isn't necessarily the global state. If a return to global state is required, call **pfFlushState()** to restore the global state. Any modification to the global state made using *libpr* functions—**pfTransparency()**, **pfDecal()**, and so on—becomes the default global state.

For best performance, set as little local pfGeoState state as possible. You can accomplish this by setting global defaults that satisfy the majority of the requirements of the pfGeoStates being drawn. By default, all pfGeoState state is inherited from the global default.

pfGeoSets and pfGeoStates

There is a special relationship between pfGeoSets and pfGeoStates. Together they completely define both geometry and graphics state. You can attach a pfGeoState to a pfGeoSet with **pfGSetGState()** to specify the appearance of geometry. Whenever the pfGeoSet is drawn with **pfDrawGSet()**, the attached pfGeoState is first applied using **pfApplyGState()**.

This combination of routines allows the application to combine geometry and state in high-performance units which are unaffected by rendering order. To further increase performance, sharing pfGeoStates among pfGeoSets is encouraged.

pfGeoState Routines

Table 10-18 lists and describes the pfGeoState routines.

Table 10-18 pfGeoState Routines

Function	Description
pfNewGState	Create a new pfGeoState.
pfCopy	Make a copy of the pfGeoState.
pfDelete	Delete the pfGeoState.
pfGStateMode	Set a specific state mode.
pfGStateVal	Set a specific state value.
pfGStateAttr	Set a specific state attribute.
pfGStateInherit	Specify which state elements are inherited from the global state.
pfApplyGState	Apply pfGeoState's non-inherited state elements to graphics.
pfGetCurGState	Return the current pfGeoState in effect.
pfGStateFuncs	Assign pre / post callbacks to pfGeoState
pfApplyGStateTable	Specify table of pfGeoStates used for indexing.

pfGeoState Structure

Figure 10-3 diagrams the conceptual structure of a pfGeoState.

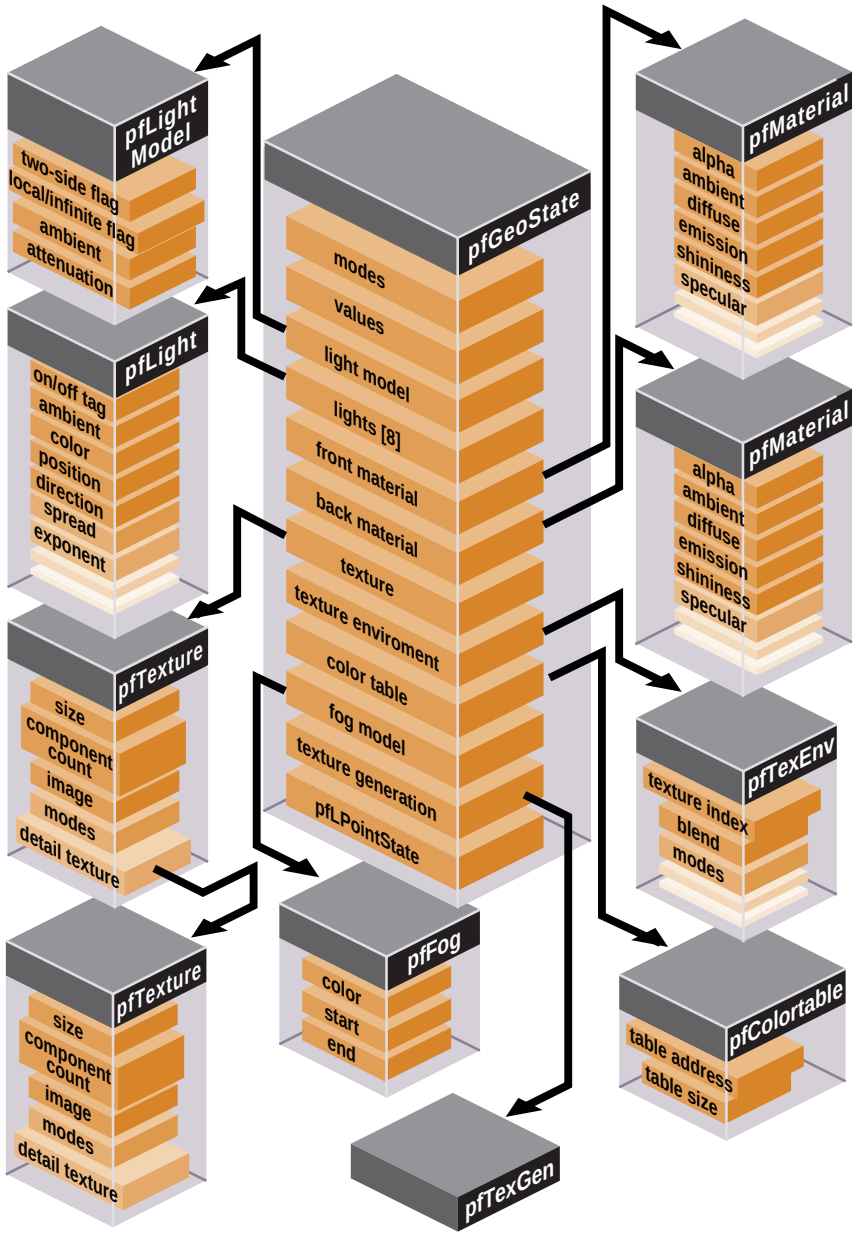


Figure 10-3 pfGeoState Structure

Windows

Rendering to the graphics hardware requires a window. A window is an allocated area of the screen with associated framebuffer resources. The X window system manages the use of shared resources amongst the different windows. Windows can be requested directly from an X window server. With a bit of interfacing, both IRIS GL and OpenGL graphics contexts can render into X windows. An X window with an IRIS GL graphics context is called a mixed model IRIS GL (GLX) window. An X window with an OpenGL graphics context is called an OpenGL/X window. An X window can only have one graphics context at a time rendering to it. IRIS GL supports a third option: pure IRIS GL windows. Pure IRIS GL windows are convenient and flexible for rendering purposes but can not be used as X windows in X applications. If, for example, you want to put your rendering window inside a larger Motif window, you will need an X window. Libpr provides utilities to shield you from the differences between the different types of windows and guide you in your dealings with the window system. Applications that use the IRIS Performer window utilities can be completely portable between IRIS GL and OpenGL and sill have the option of using pure IRIS GL windows if desired when running in IRIS GL. You'll be able to use your windows in X applications, or direct your rendering to a pre-created window. The *libpr* windowing support centers around the pfWindow.

pfWindows are structures for managing any of the different kinds of windows and associated pfState. pfWindows provide an efficient windowing interface between your application and the window system. pfWindows shield you from the functional and performance differences between the different graphics libraries and the different window system interfaces and allow you to configure, manipulate, and query IRIS GL, IRIS GL mixed model (GLX), and OpenGL/X windows through a GL and window system independent interface. pfWindows also keep track of your graphics state: they include a pfState which is automatically initialized when you open a window, and switched for you when you change windows.

IRIS Performer will automatically configure and initialize your window so that it will be ready to start rendering efficiently. In the simplest case, pfWindows make creating a graphics application that can run on any Silicon Graphics machine with IRIS GL or OpenGL a snap. pfWindows do not limit your ability to configure any part or all of your windowing environment

yourself; you can use the *libpr* pfWindows to manage your GL windows even if you create and configure the actual windows yourself.

A pfWindow structure is created with **pfNewWin()**. It can then be immediately opened with **pfOpenWin()**. Example 10-7 shows the most basic pfWindow operations in *libpr* program: to open and clear a pfWindow and swap front and back color buffers.

Example 10-7 Opening a pfWindow

```
int main (void)
{
     pfWindow *win;
     /* Initialize Performer */
     pfInit();
     pfInitState(NULL);

     /* Create and open a Window */
     win = pfNewWin(NULL);
     pfWinName(win, “Hello from IRIS Performer”);
     pfOpenWin();

     /* Rendering loop */
     while (1)
     {
         /* Clear to black and max depth */
         pfClear(PFCL_COLOR | PFCL_DEPTH, NULL);
         ...
         pfSwapWinBuffers(win);
     }
}
```

The pfWindow in Example 10-7 will have the following configuration:

Window system interface
: OpenGL windows will be an X window using the OpenGL / X interface. IRIS GL windows will be pure IRIS GL.

Screen:
: The pfWindow will open a window on the screen specified by the DISPLAY environment variable, or else on screen 0.

Position and size:
: The position and size will be undefined and the window will come up as a rubber-bend for the user to place and stretch.

Framebuffer configuration:
: The window will be doublebuffered RGBA with depth and stencil buffers allocated. The size of these buffers will depend on the available resources of the current graphics hardware platform. GLX and OpenGL/X windows will also have multisample buffers allocated if they are available on current hardware platform.

Libpr state:
: A pfState will be created and initialized with all modes disabled and no attributes set.

Graphics state:
: The pfWindow will be in RGBA color mode with subpixel vertex positioning, depth testing and viewport clipping enabled. The Viewing projection will be a two-dimensional one-one orthographic mapping from eye coordinates to window coordinates with distances to near and far clipping planes -1 and 1, respectively. The model matrix will be the current matrix and will be initialized to the identity matrix.

Typically, pfWindows go through a bit more initialization than that of Example 10-7. The type of pfWindow type, set with **pfWinType()** is a bitmask that selects the window system interface and the type of rendering window. The default window type is a normal graphics rendering window and is pure IRIS GL under IRIS GL operation and X under OpenGL. operation. Table 10-19 lists the possible selectors that can be OR-ed together for specification of the window type.

Table 10-19 **pfWinType()** Tokens

PFWIN_TYPE_ Bitmask Token	Description
X	Window will be an X window, as opposed to a pure IRIS GL window. This only has effect under IRIS GL operation.
STATS	Window will have framebuffer resources to accommodate hardware statistics modes. This type cannot be combined with PFWIN_TYPE_OVERLAY or PFWIN_TYPE_NOPORT.

Table 10-19 (continued) **pfWinType()** Tokens

PFWIN_TYPE_ Bitmask Token	Description
OVERLAY	Window will have only overlay planes for rendering. This type cannot be combined with PFWIN_TYPE_STATS or PFWIN_TYPE_NOPORT.
NOPORT	Window will have a graphics context but no physical window or graphics or framebuffer rendering resources and will not be placed on the screen. This token should not be used in combination with any other type token.

The selection of screen can be done explicitly with **pfWinScreen()**, or implicitly by opening a connection to the window system using **pfOpenScreen()** with the desired screen as the default screen. A window system connection can communicate with any screen on the system; the default screen only determines the screen for windows that do not have a screen explicitly set for them. Only one window system connection should be opened for a process. See "Communicating with the Window System" later in this section for details on efficient interaction with the window system.

The position and / or size, is set with **pfWinOriginSize()**. If the x and y components of the origin are (-1), the window will open with position undefined for the user to place. If the x or y components of the size are (-1), the window will open with both position and size undefined (the default) for the user to place and stretch. The X window manager may override negative origins and place the window at (0,0). If the window is already opened when **pfWinOriginSize()** is called, the window will be reconfigured to the specified origin and size upon the next **pfSelectWin()**. Similarly, **pfWinFullScreen()** will cause a window to open as full screen or to become full screen upon the next call to **pfSelectWin()**. A full screen window will have its border automatically removed so that the drawing area truly gets the full rendering surface. The routines for querying the position and size work a bit differently than the pattern established by the rest of *libpr* get and set pairs of routines. This is because a user may change the origin or size independently of the program and under certain conditions, querying the true current X window size and origin can be expensive. **pfGetWinOrigin()** and **pfGetWinSize()** will always be fast and will return the last explicitly set origin and size, such as by **pfOpenWin()**, **pfWinOriginSize()**, or

pfWinFullScreen(). If the window origin or size has been changed, but not through a pfWindow routine, the values returned by **pfGetWinOrigin()** and **pfGetWinSize()** may not be correct. **pfGetWinCurOriginSize()** will return an accurate size and origin relative to the pfWindow parent. For pure IRIS GL windows, this will also be reasonably fast; however, for X windows, it will be expensive and should not be done in real-time situations. The parent of an IRIS GL window is always the screen, but not so with X windows. **pfGetWinCurScreenOriginSize()** will return the size and the screen-relative origin of the pfWindow. If the pfWindow is an X window, this command will be quite expensive and is not recommended accept for rare use or initialization purposes.

pfPipeWindows, discussed in Chapter 4, “Setting Up the Display Environment,” take advantage of the multiprocessed *libpf* environment to always be able to return an accurate window size and origin relative to the window parent. A process separate from the rendering process is by the window manager of changes in the pfPipeWindow’s size in an efficient manner without impacting the window system or the rendering process. However, even for pfPipeWindows, getting a screen-relative origin can be an expensive operation.

Hint: Users are strongly encouraged to write programs that are window-relative and do not depend on knowing the current exact location of a window relative to its parent or screen.

Configuring the framebuffer of a pfWindow

IRIS Performer provides a default framebuffer configurations for the current graphics hardware platform for the standard window types: normal rendering, statistics (stats), and overlay. You may want to define your own framebuffer configuration, such as single-buffered, stereo, etc. You can use utilities in *libpr* to help you with this task, or create your own framebuffer configuration structure with X utilities, or even create the window yourself and apply it to the pfWindow. **pfOpenWin()** will respect any specified framebuffer configuration. Additionally, **pfOpenWin()** uses any window or graphics context that is assigned to it and only creates what is undefined.

pfWinFBConfigAttrs() can be used to specify an array of framebuffer attribute tokens listed in Table 10-20. The tokens are exactly like the OpenGL / X tokens and the same attribute list can be used for all window types: pure IRIS GL, mixed model IRIS GL, and OpenGL / X windows. Note that if an attribute array is specified, the tokens modify configuration with no attributes set, not the default IRIS Performer framebuffer configuration.

Table 10-20 pfWinFBConfigAttrs() Tokens

PFFB_ Token	Value	Description
BUFFER_SIZE	integer > 0	The size of the color index buffer
LEVEL	integer > 0	The color plane level: normal color planes have level = 0 overlay color planes have level > 0 underlay color planes have level < 0 There may be only one or no levels for overlay and underlay color planes on some graphics hardware configurations.
RGBA	Boolean: true if present	Use RGBA color planes (instead of color index)
DOUBLEBUFFER	Boolean: true if present	Use double-buffered color buffers
STEREO	Boolean: true if present	Allocate left and right stereo color buffers (allocates back left and back right if DOUBLEBUFFER is specified.
AUX_BUFFER	integer > 0	Number of additional color buffers to allocate
RED_SIZE GREEN_SIZE BLUE_SIZE ALPHA_SIZE	integer > 0	Minimum number of bits color for components R, G, and B will all be the same and be the maximum specified. Alpha may be different.
DEPTH_SIZE	integer > 0	Number of bits in the depth buffer

Table 10-20 (continued) **pfWinFBConfigAttrs()** Tokens

PFFB_ Token	Value	Description
STENCIL	integer > 0	Number of bits allocated for stencil. One is used by pfDecal rendering and three or four are used by the hardware fill statistics in pfStats.
ACCUM_RED_SIZE ACCUM_GREEN_SIZE ACCUM_BLUE_SIZE ACCUM_ALPHA_SIZE	integer > 0	Number of bits per RGBA component for the accumulation color buffer.
USE_GL	Boolean: true if present	Accepted for compatibility with X routines. Has no effect.

You may get back a framebuffer configuration that is better than the one you requested. IRIS Performer will give you back the maximum framebuffer configuration that meets your request that will not add any serious performance degradations. There are specific machine dependent instances where when possible, for performance reasons, we do limit the framebuffer configuration. See the **pfChooseWinFBConfig()** reference page for the specific details.

If you desire more control over the exact framebuffer configuration of your pfWindow, you have several options. For pure IRIS GL windows you can make GL framebuffer configuration calls, such as **RGBsize()**, **zbsize()**, and **mssize()**, directly. You can tell IRIS Performer to not do any window configuration by setting an empty attribute array. For X windows you can provide the appropriate framebuffer description for the current GL operation to the pfWindow using **pfWinFBConfig()**. X uses *visuals* to describe available framebuffer configurations. You can select the visual for your window and set it on the pfWindow with **pfWinFBConfig()**. XVisualInfo pointer with **XGetVisualInfo()** will return a list of all visuals on the system and you can search through them to find the appropriate configuration.

libpr also offers utilities for creating framebuffer configurations (pfFBConfig) independently of a pfWindow. **pfChooseFBConfig()** takes an attribute array of tokens from Table 10-20 and will return a pfFBConfig structure that can be used with your pfWindows, or with X Windows created outside of *libpr*, such as with Motif.

You can use **pfQuerySys()** to query particular framebuffer resources in the current hardware configuration and then use **pfQueryWin()** to query your resulting framebuffer configuration.

There is a special utility for supporting mixed model IRIS GL (GLX) windows. GLX windows use a special attribute array returned by **GLXgetconfig()** and expected by **GLXlink()** for creating a framebuffer configuration and graphics window. You can set and get this special GL-dependent attribute array with **pfWinFBConfigData()** and **pfGetWinFBConfigData()**, respectively. This configuration array is useful for hooking up GLX windows with X windows from other toolkits or with Motif. Under OpenGL operation, **pfWinFBConfigData()** just expects a configuration attribute array appropriate for **glXChooseVisual()** or **pfChooseWinFBConfig()**.

pfWindows and GL Windows

libpr allows you direct access to the GL and X window handles, or to create your own windows and set them on the pfWindow.

You can create your own windows (and / or in the case of OpenGL / X, graphics contexts) and set them on the pfWindow. You can then call **pfOpenWin()** to make sure everything is hooked up correctly, apply any specified origin and size, and to initialize your IRIS Performer state. Under pure IRIS GL operation, a window and a graphics context are the same thing. A pure IRIS GL window is created with the **winopen()** command and can be assigned to the pfWindow with **pfWinWSDrawable()**, or **pfWinGLCxt()**.

pfOpenWin() will automatically call **pfInitGfx()** and will automatically create a new pfState for your window. If you have your own window management and do not call **pfOpenWin()** then you should definitely call **pfInitGfx()** to initialize the window's graphics state for IRIS Performer rendering and you will also need to call **pfNewState()** to create a pfState for IRIS Performer's state management.

For X windows, IRIS Performer maintains two windows and a graphics context. The top level X window is the one that placed on the screen and is the one that you should use in your application for selecting X events. This top level window is very lightweight and has minimal resources allocated to it. IRIS Performer then maintains a separate X window that is a child of the parent X window and is the one that is attached to the graphics context. This allows you to select different framebuffer resources for the same drawing area by just selecting a different graphics window and graphics context pair for the parent X window. pfWindows directly support this functionality and this is discussed in the next section, “Manipulating a pfWindow“. Finally, with OpenGL, you may choose to draw to a different X Drawable than a window. X windows are created with the X function **XCreateWindow()**. Mixed model IRIS GL windows have an X window serve as both the graphics drawable and the GL context. OpenGL graphics contexts are created with **glXCreateContext()**. The parent X Window can be set with **pfWinWSWindow()**, the graphics window or X Drawable is set with **pfWinWSDrawable()**, and the OpenGL graphics context is set with **pfWinGLCxt()**. For compatibility between GLs, IRIS Performer defines the following GL and Window System independent types defined in Table 10-21 If you create your own window but want to use **pfQueryWin()** you must also provide the framebuffer configuration information with **pfWinFBConfig()** and **pfWinFBConfigData()** for OpenGL and GLX respectively. **pfQueryWin()** uses the internally stored visual, and in the case of GLX, the attribute array given to the GLX call **GLXlink()**. .

Table 10-21 Window System Types

pfWS Type	X Type	pfWindow Set/Get Routine
pfWSWindow	Window	**pfWinWSWindow()** **pfGetWinWSWindow()**
pfWSDrawable	Drawable	**pfWinWSDrawable()** **pfGetWinWSDrawable()**
pfGLContext	IRIS GL: int OpenGL: GLXContext	**pfWinGLCxt()** **pfGetWinGLCxt()**
pfFBConfig	XVisualInfo*	**pfWinFBConfig()** **pfGetWinFBConfig()**
pfWSConnection	Display	**pfGetCurWSConnection()**

Manipulating a pfWindow

Windows are opened with **pfOpenWin()** and closed with **pfCloseWin()**. When a window is closed, its graphics context is deleted. If you have multiple windows, you select the window to draw to with **pfSelectWin()**. Windows can be dynamically resized with the same sizing commands discussed previously for setting up the window.

IRIS Performer supports multiple framebuffer configurations for the same drawing area in a GL independent fashion with alternate configuration windows. An IRIS Performer alternate configuration window has the same window parent (**pfWinWSWindow()**) but may have a different drawable and graphics context. There are standard alternate configuration windows for overlay and statistics windows that can be automatically created upon demand. For pure IRIS GL, alternate configuration windows must have their graphics context be the same as the base window. The rest of this section assumes the use of X windows in either GLX or OpenGL/X.

An alternate configuration window is created as a full pfWindow and is an alternate configuration window by virtue of being given to a base window in a pfList of alternate configuration windows, or being directly assigned as one of the standard alternate configuration windows with either of **pfWinOverlayWin()** or **pfWinStatsWin()**. A pfWindow may be an alternate configuration window of only one base window at a time; alternate configuration windows may not be instanced between base windows. The sharing of window attributes between alternate configuration windows, such as the parent X window and GL objects (for OpenGL windows), must be set with **pfWinShare()** on the base window and applied to the alternate configuration windows with **pfAttachWin()**. You select the desired alternate configuration window to draw into with **pfWinIndex()** and provide an index into your alternate configuration window list or one of the standard indices (PFWIN_GFX_WIN, PFWIN_OVERLAY_WIN, or PFWIN_STATS_WIN). PFWIN_GFX_WIN is the default window index and selects the base window. If the alternate configuration window has not been opened, it will be opened automatically upon being selected for rendering. Example 10-9 demonstrates creating a pfWindow using the default overlay window. The graphics drawable and graphics context of an alternate configuration window of a pfWindow can be closed with **pfCloseWinGL()**. This can be called on the base window, in which case the active alternate configuration window's GL window and context will be closed, or it can be called on the alternate configuration window pfWindow directly. The main

parent window will remain on the screen and a new alternate configuration window can be applied to it or **pfOpenWin()** can be called to create a new graphics window and context.

There are some modes you can set that can effect the general look and behavior of your window and alternate configuration windows. These boolean modes can be individually set and changed at any time with **pfWinMode()** and the tokens in Table 10-22.

Table 10-22 **pfWinMode()** Tokens

PFWIN_ Token	Description
NOBORDER	Window will be without normal window system border
HAS_OVERLAY	Overlay alternate configuration window will be managed by the pfWindow. **pfOpenWin()** will automatically create an overlay window if one has not already been set. **pfWinIndex(win, PFWIN_OVERLAY_WIN)** will also automatically create and open an overlay window if one has not already been set. This mode only has effect for X windows.
HAS_STATS	Statistics alternate configuration window will be managed by the pfWindow. **pfOpenWin()** will automatically create a statistics window if one has not already been set. **pfWinIndex(win, PFWIN_OVERLAY_WIN)** will also automatically create and open a statistics window if one has not already been set and if the current window cannot support statistics.This mode only has effect for X windows.
AUTO_RESIZE	The graphics window and active alternate configuration windows are automatically resized to match the parent **pfWinWSWindow()**. This mode is enabled by default and only has effect for X windows.
ORIGIN_LL	The origin of the pfWindow, for placement purposes, will be the lower-left corner. X uses the upper left corner as the origin and pure IRIS GL uses the lower-level. This mode is enabled by default.
EXIT	The application will receive a DeleteWindow message upon selection of the “Exit” from the window system menu on the window border. This mode only has effect for X windows.

Communicating with the Window System

You can communicate with a local or remote window server by means of a window system connection, a pfWSConnection (in X, also known as a Display connection). You can use your pfWSConnection for selecting X events for your window, as is demonstrated in Example 10-11.

Libpr offers several utilities for creating a connection to a window server. A given connection can communicate with any screen managed by that window server so usually a process only needs one connection. A process should not share the connection of another process, so you will need a connection per process. Typically, there is exactly one window server on a machine but that is not required. *Libpr* maintains a pfWSConnection for the current process. By default, this connection obeys the setting of the DISPLAY environment variable which can point to a window server on a local or a remote machine. The current connection can be requested with **pfGetCurWSConnection()**, and can be set with **pfSelectWSConnection()**. It is recommended that whenever possible, this connection be used to limit the total number of open connections. **pfOpenScreen()** is a convenient mechanism for opening a connection with a specified default screen. **pfOpenWSConnection()** allows you to specify the exact name specifying the desired target for the connection. Both **pfOpenScreen()** and **pfOpenWSConnection()** allow you to specify if you would like the new connection to automatically be made the current libpr pfWSConnection; this is recommended.

More pfWindow Examples

Example 10-8 demonstrates creating a window that will support a desired fill statistics configuration. This example is taken from the sample program */usr/share/Performer/src/pguide/libpr/C/fillstats.c*. Statistics are the topic of Chapter 12, “Statistics.”

Example 10-8 Creating a Statistics Window

```
int main (void)
{
    pfWindow *win;
    int bits, sten;
    /* Initialize Performer */
    pfInit();
```

```
    pfInitState(NULL);

     /* set up number of bits needed for stats before
      * window is made so that it will have the right number
      */
     pfQuerySys(PFQSYS_MAX_STENCIL_BITS,&sten);
     bits = PF_MIN2(4, sten);
     pfStatsHwAttr(PFSTATSHW_FILL_DCBITS, bits);

     /* Initialize GL */
     win = pfNewWin(NULL);
     pfWinOriginSize(win, 100, 100, 500, 500);
     pfWinName(win, “Iris Performer”);
     pfWinType(win, PFWIN_TYPE_X | PFWIN_TYPE_STATS );
     pfOpenWin(win);
     ....
}
```

Example 10-9 demonstrates the creation of a window with a default overlay window.

Example 10-9 Using the Default Overlay Window

```
int main (void)
{
    pfWindow *win, *over;
    /* Initialize Performer */
    pfInit();
    pfInitState(NULL);

    /* Initialize the window. */
     win = pfNewWin(NULL);
     pfWinOriginSize(win, 100, 100, 500, 500);
     pfWinName(win, “Iris Performer”);
     pfWinType(win, PFWIN_TYPE_X);
     pfWinMode(win, PFWIN_HAS_OVERLAY, 1);
     pfOpenWin(win);
     /* First select and draw into the overlay window */
     pfWinIndex(win, PFWIN_OVERLAY_WIN);
     /* Select causes the index to be applied */
     pfSelectWin(win);
     ...
     /* Then select the main gfx window */
     pfWinIndex(win, PFWIN_GFX_WIN);
     pfSelectWin(win);
```

```
    ...
}
```

Example 10-10 demonstrates creating a custom overlay window and is taken from the sample program */usr/share/Performer/src/pguide/libpr/C/winfbconfig.c*.

Example 10-10 Creating a Custom Overlay Window

```
static int OverlayAttrs[] = {
 PFFB_LEVEL, 1, /* Level 1 indicates overlay visual */
 PFFB_BUFFER_SIZE, 8,
 None,
};

int main (void)
{
    pfWindow *win, *over;
    /* Initialize Performer */
    pfInit();
    pfInitState(NULL);

    /* Initialize the window. */
     win = pfNewWin(NULL);
     pfWinOriginSize(win, 100, 100, 500, 500);
     pfWinName(win, “Iris Performer”);
     pfWinType(win, PFWIN_TYPE_X);
     pfWinMode(win, PFWIN_HAS_OVERLAY, 1);

     over = pfNewWin(NULL);
     pfWinName(over, “Iris Performer Overlay”);
     pfWinType(over, PFWIN_TYPE_X | PFWIN_TYPE_OVERLAY);
     /* See if we can get the desired overlay visual */
     if (!(pfChooseWinFBConfig(over, OverlayAttrs)))
         pfNotify(PFNFY_NOTICE, PFNFY_PRINT,
            “pfChooseWinFBConfig failed for OVERLAY win”);
     pfOpenWin(win);
     /* First select and draw into the overlay window */
     pfWinIndex(win, PFWIN_OVERLAY_WIN);
     /* Select causes the index to be applied */
     pfSelectWin(win);
     ...
     /* Then select the main gfx window */
     pfWinIndex(win, PFWIN_GFX_WIN);
     pfSelectWin(win);
     ...
```

```
}
```

Example 10-11 demonstrates the selection of X input events on a pfWindow. This example is taken from */usr/share/Performer/src/pguide/libpr/C/hlcube.c*. See the */usr/share/Performer/src/pguide/libpf/C/complex.c* sample program for a detailed example of using either GL or forked X input on pfWindows.

Example 10-11 pfWindows and X Input

```
    pfWSConnection Dsp;

void main (void)
{
    pfWindow *win;
    pfWSWindow xwin;

    /* Initialize Performer */
    pfInit();
    pfInitState(NULL);

    /* Initialize the window. */
     win = pfNewWin(NULL);
     pfWinOriginSize(win, 100, 100, 500, 500);
     pfWinName(win, “Iris Performer”);
     pfWinType(win, PFWIN_TYPE_X);
     pfOpenWin(win);
     ...
     /* set up X input event handling on pfWindow */
     Dsp = pfGetCurWSConnection();
     xwin = pfGetWinWSWindow(win);
     XSelectInput(Dsp, xwin, KeyPressMask );
     XMapWindow(Dsp, xwin);
     XSync(Dsp,FALSE);
     ...
     do_events(win);
}
static void
do_events(pfWindow *win)
{
    while (1) {
    while (XPending(dsp))
    {
         XEvent event;
         XNextEvent(Dsp, &event);
         switch (event.type)
```

```
        {
        case KeyPress:
         ....
         }
    }
}
```

libpr Sample Code

Example 10-12 shows how to make a colored cube. This example is derived from source code in */usr/share/Performer/src/pguide/libpr/C/colorcube.c*.

Example 10-12 Constructing a Colored Cube With *libpr*

```
/*
 * colorcube.c - routine for constructing colored cube geoset
 */

#include <Performer/pr.h>
#include <stdlib.h>

#defineCUBE_SIZE1.0f

pfGeoSet*
MakeColorCube(void *arena)
{
    pfGeoSet *gset;
    pfGeoState *gst;

    pfVec4 *scolors;
    pfVec3 *snorms, *scoords;
    ushort *nindex, *vindex, *cindex;

    /*
     * Data arrays to be passed to pfGSetAttr should be
     * allocated from heap memory...
     */
     scolors = (pfVec4 *)pfMalloc(4 * sizeof(pfVec4), arena);
     pfSetVec4(scolors[0], 1.0f, 0.0f, 0.0f, 0.5f);
     pfSetVec4(scolors[1], 0.0f, 1.0f, 0.0f, 0.5f);
     pfSetVec4(scolors[2], 0.0f, 0.0f, 1.0f, 0.5f);
     pfSetVec4(scolors[3], 1.0f, 1.0f, 1.0f, 0.5f);
```

```
snorms = (pfVec3 *)pfMalloc(6 * sizeof(pfVec3), arena);
pfSetVec3(snorms[0], 0.0f, 0.0f, 1.0f);
pfSetVec3(snorms[1], 0.0f, 0.0f, -1.0f);
pfSetVec3(snorms[2], 0.0f, 1.0f, 0.0f);
pfSetVec3(snorms[3], 0.0f, -1.0f, 0.0f);
pfSetVec3(snorms[4], 1.0f, 0.0f, 0.0f);
pfSetVec3(snorms[5], -1.0f, 0.0f, 0.0f);

scoords = (pfVec3 *)pfMalloc(8 * sizeof(pfVec3), arena);
pfSetVec3(scoords[0], -CUBE_SIZE, -CUBE_SIZE,
                       CUBE_SIZE);
pfSetVec3(scoords[1], CUBE_SIZE, -CUBE_SIZE, CUBE_SIZE);
pfSetVec3(scoords[2], CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
pfSetVec3(scoords[3], -CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
pfSetVec3(scoords[4], -CUBE_SIZE, -CUBE_SIZE,
                      -CUBE_SIZE);
pfSetVec3(scoords[5], CUBE_SIZE, -CUBE_SIZE,
                     -CUBE_SIZE);
pfSetVec3(scoords[6], CUBE_SIZE, CUBE_SIZE, -CUBE_SIZE);
pfSetVec3(scoords[7], -CUBE_SIZE, CUBE_SIZE,
                      -CUBE_SIZE);

nindex = (ushort *)pfMalloc(6 * sizeof(ushort), arena);
nindex[0] = 0; nindex[1] = 5; nindex[2] = 1;
nindex[3] = 4; nindex[4] = 2; nindex[5] = 3;

vindex = (ushort *)pfMalloc(24 * sizeof(ushort), arena);
vindex[0]  = 0; vindex[1]  = 1; /* front */
vindex[2]  = 2; vindex[3]  = 3;
vindex[4]  = 0; vindex[5]  = 3; /* left */
vindex[6]  = 7; vindex[7]  = 4;
vindex[8]  = 4; vindex[9]  = 7; /* back */
vindex[10] = 6; vindex[11] = 5;
vindex[12] = 1; vindex[13] = 5; /* right */
vindex[14] = 6; vindex[15] = 2;
vindex[16] = 3; vindex[17] = 2; /* top */
vindex[18] = 6; vindex[19] = 7;
vindex[20] = 0; vindex[21] = 4; /* bottom */
vindex[22] = 5; vindex[23] = 1;

cindex = (ushort *)pfMalloc(24 * sizeof(ushort), arena);
cindex[0] = 0; cindex[1] = 1;
cindex[2] = 2; cindex[3] = 3;
cindex[4] = 0; cindex[5] = 1;
cindex[6] = 2; cindex[7] = 3;
```

```
    cindex[8] = 0; cindex[9] = 1;
    cindex[10] = 2; cindex[11] = 3;
    cindex[12] = 0; cindex[13] = 1;
    cindex[14] = 2; cindex[15] = 3;
    cindex[16] = 0; cindex[17] = 1;
    cindex[18] = 2; cindex[19] = 3;
    cindex[20] = 0; cindex[21] = 1;
    cindex[22] = 2; cindex[23] = 3;

    /* Allocate the pfGeoSet out of the same arena. */
    gset = pfNewGSet(arena);

    /*
     * set the coordinate, normal and color arrays
     * and their corresponding index arrays
     */
    pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX,
               scoords, vindex);
    pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM,
               norms, nindex);
    pfGSetAttr(gset, PFGS_COLOR4, PFGS_PER_VERTEX,
               scolors, cindex);
    pfGSetPrimType(gset, PFGS_QUADS);
    pfGSetNumPrims(gset, 6);

    /*
     * create a new geostate from shared memory,
     * disable texturing and enable transparency
     */
    gst = pfNewGState(arena);
    pfGStateMode(gst, PFSTATE_ENTEXTURE, 0);
    pfGStateMode(gst, PFSTATE_TRANSPARENCY, 1);

    pfGSetGState(gset, gst);
    return gset;
}
```

Example 10-13 demonstrates construction of a textured cube. This example can be found in the file */usr/share/Performer/src/pguide/libpr/C/texcube.c*.

Example 10-13 Constructing a Textured Cube With *libpr*

```
/*
 * texcube.croutine for constructing a textured cube geoset
 */

#include <Performer/pr.h>

#define CUBE_SIZE1.0f

pfGeoSet*
MakeTexCube(void *arena)
{
     pfGeoSet *gset;
     pfGeoState *gst;
     pfTexture *tex;
     pfTexEnv *tenv;
     pfVec3 *verts, *norms;
     pfVec2 *tcoords;
     ushort *vindex, *nindex, *tindex;

     /*
     * Data arrays to be passed to pfGSetAttr should be
     * allocated from heap memory...
     /
     verts = (pfVec3 *)pfMalloc(8 * sizeof(pfVec3), arena);
     pfSetVec3(verts[0], -CUBE_SIZE, -CUBE_SIZE, CUBE_SIZE);
     pfSetVec3(verts[1], CUBE_SIZE, -CUBE_SIZE, CUBE_SIZE);
     pfSetVec3(verts[2], CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
     pfSetVec3(verts[3], -CUBE_SIZE, CUBE_SIZE, CUBE_SIZE);
     pfSetVec3(verts[4], -CUBE_SIZE, -CUBE_SIZE, -CUBE_SIZE);
     pfSetVec3(verts[5], CUBE_SIZE, -CUBE_SIZE, -CUBE_SIZE);
     pfSetVec3(verts[6], CUBE_SIZE, CUBE_SIZE, -CUBE_SIZE);
     pfSetVec3(verts[7], -CUBE_SIZE, CUBE_SIZE, -CUBE_SIZE);

     vindex = (ushort *)pfMalloc(24 * sizeof(ushort), arena);
     vindex[0]  = 0; vindex[1]  = 1;  /* front */
     vindex[2]  = 2; vindex[3]  = 3;
     vindex[4]  = 0; vindex[5]  = 3;  /* left */
     vindex[6]  = 7; vindex[7]  = 4;
     vindex[8]  = 4; vindex[9]  = 7;  /* back */
     vindex[10] = 6; vindex[11] = 5;
```

```
vindex[12] = 1; vindex[13] = 5;  /* right */
vindex[14] = 6; vindex[15] = 2;
vindex[16] = 3; vindex[17] = 2;  /* top */
vindex[18] = 6; vindex[19] = 7;
vindex[20] = 0; vindex[21] = 4;  /* bottom */
vindex[22] = 5; vindex[23] = 1;

norms = (pfVec3 *)pfMalloc(6 * sizeof(pfVec3), arena);
pfSetVec3(norms[0], 0.0f, 0.0f, 1.0f);
pfSetVec3(norms[1], 0.0f, 0.0f, -1.0f);
pfSetVec3(norms[2], 0.0f, 1.0f, 0.0f);
pfSetVec3(norms[3], 0.0f, -1.0f, 0.0f);
pfSetVec3(norms[4], 1.0f, 0.0f, 0.0f);
pfSetVec3(norms[5], -1.0f, 0.0f, 0.0f);

nindex = (ushort *)pfMalloc(6 * sizeof(ushort), arena);
nindex[0] = 0; nindex[1] = 5; nindex[2] = 1;
nindex[3] = 4; nindex[4] = 2; nindex[5] = 3;

tcoords = (pfVec2 *)pfMalloc(4 * sizeof(pfVec2), arena);
pfSetVec2(tcoords[0], 0.0f, 0.0f);
pfSetVec2(tcoords[1], 1.0f, 0.0f);
pfSetVec2(tcoords[2], 1.0f, 1.0f);
pfSetVec2(tcoords[3], 0.0f, 1.0f);

tindex = (ushort *)pfMalloc(24 * sizeof(ushort), arena);
tindex[0]  = 0; tindex[1]  = 1;
tindex[2]  = 2; tindex[3]  = 3;
tindex[4]  = 0; tindex[5]  = 1;
tindex[6]  = 2; tindex[7]  = 3;
tindex[8]  = 0; tindex[9]  = 1;
tindex[10] = 2; tindex[11] = 3;
tindex[12] = 0; tindex[13] = 1;
tindex[14] = 2; tindex[15] = 3;
tindex[16] = 0; tindex[17] = 1;
tindex[18] = 2; tindex[19] = 3;
tindex[20] = 0; tindex[21] = 1;
tindex[22] = 2; tindex[23] = 3;

/* Allocate the pfGeoSet out of the same arena */
gset = pfNewGSet(arena);

/*
 * set the coordinate, normal and color arrays
 * and their corresponding index arrays
```

```
     */
    pfGSetAttr(gset, PFGS_COORD3, PFGS_PER_VERTEX,
                     verts, vindex);
    pfGSetAttr(gset, PFGS_NORMAL3, PFGS_PER_PRIM,
                     norms, nindex);
    pfGSetAttr(gset, PFGS_TEXCOORD2, PFGS_PER_VERTEX,
                     tcoords, tindex);
    pfGSetPrimType(gset, PFGS_QUADS);
    pfGSetNumPrims(gset, 6);

    /*
     * create a geostate from the arena, enable
     * texturing (in case that's not the default), and
     * set the geostate for this geoset
     */
    gst = pfNewGState(arena);
    pfGStateMode(gst, PFSTATE_ENTEXTURE, 1);
    pfGSetGState(gset, gst);

    /*
     * create a new texture from shared memory,
     * load a texture file and add texture to geostate
     */
    tex = pfNewTex(arena);
     pfLoadTexFile(tex, "brick.rgba");
     pfGStateAttr(gst, PFSTATE_TEXTURE, tex);

     /* Create a new texture environment from the arena. */
     tenv = pfNewTEnv(arena);
     /* Decal the texture since the geoset has no color to
      * modulate.
      */
     pfTEnvMode(tenv, PFTE_DECAL);
     /* set the texture environment for this pfGeoState *./
     pfGStateAttr(gst, PFSTATE_TEXENV, tenv);

     return gset;
}
```

Managing Nongraphic System Tasks

This section describes routines that manage nongraphic tasks such as clocks, memory, I/O, and error handling.

Clocks

IRIS Performer provides clock support for timing operations.

High-Resolution Clock

IRIS Performer provides access to a high-resolution clock that reports elapsed time in seconds. To start a clock, call **pfInitClock()** with the initial time in seconds—usually 0.0—as the parameter. Subsequent calls to **pfInitClock()** reset the time to whatever value you specify. To read the time, call **pfGetTime()**. This function returns a double-precision floating point number representing the seconds elapsed from initialization added to the latest reset value.

The resolution of the clock depends on your system type and configuration. In most cases, the resolved time interval is under a microsecond, and so is much less than the time required to process the **pfGetTime()** call itself. Exceptions are older Power Series™ machines with IO2 boards and some Personal Iris™ computers. You can use the IRIX *hinv* command to see whether your Power Series system has an IO2 board or an IO3 board installed. Onyx, Crimson™, Indigo^2™, Indigo®, and Indy™ systems all provide submicrosecond resolution. On a machine that uses a fast hardware counter, the first invocation of **pfInitClock()** forks off a process that periodically wakes up and checks the counter for wrapping. This additional process can be suppressed using **pfClockMode()**.

If IRIS Performer cannot find a fast hardware counter to use, it defaults to the time-of-day clock, which typically has a resolution between one and ten milliseconds. This clock resolution can be improved by using fast timers. See the *ftimer*(1) reference page for more information on fast timers.

By default, processes forked after the first call to **pfInitClock()** share the same clock and will all see the results of any subsequent **pfInitClock()**. All such processes receive the same time.

Unrelated processes can share the same clock by calling **pfClockName()** with a clock name before calling **pfInitClock()**. This provides a way to name and reference a clock. By default, unrelated processes don't share clocks.

Video Refresh Counter (VClock)

The video refresh counter (VClock) is a counter that increments once for every vertical retrace interval. There is one VClock per system. In systems where multiple graphics pipelines are present, but not genlocked (synchronized, see the **setmon(3)** reference page), screen 0 is used as the source for the counter. A process can be blocked until a certain count, or the count modulo some value (usually desired number of video fields per frame) is reached.

Table 10-23 lists and describes the pfVClock routines.

Table 10-23 pfVClock Routines

Function	Action
pfInitVClock	Initialize the clock to a value.
pfGetVClock	Get the current count.
pfVClockSync	Block the calling process until a count is reached.

When using **pfVClockSync()**, the calling routine is blocked until the current count modulo *rate* is *offset*. The VClock functions can be used to synchronize several channels or pipelines.

Memory Allocation

You can use IRIS Performer memory-allocation functions to allocate memory from the heap, from shared memory, and from datapools.

Table 10-24 lists and describes the IRIS Performer shared memory routines.

Table 10-24 Memory Allocation Routines

Function	Action
pfInitArenas	Create arenas for shared memory and semaphores.
pfSharedArenaSize	Specify the size of a shared memory arena.
pfGetSharedArena	Get the shared memory arena pointer.
pfGetSemaArena	Get the shared semaphore/lock arena pointer.
pfMalloc	Allocate from an arena or the heap.
pfFree	Release memory allocated with pfMalloc().

Allocating Memory With pfMalloc()

pfMalloc() can allocate memory either from the heap or from a shared memory arena. Multiple processes can access memory allocated from a shared memory arena, whereas memory allocated from the heap is visible only to the allocating process. Pass a shared-memory arena pointer to **pfMalloc()** to allocate memory from the given arena. **pfGetSharedArena()** returns the pointer for the arena allocated by **pfInitArenas()**, or NULL if the given memory was allocated from the heap. Alternately, an application can create its own shared memory arena; see the **acreate**(3P) reference page for information on how to create an arena.

To allocate memory from the heap, pass NULL to **pfMalloc()** instead of an arena pointer.

Under normal conditions **pfMalloc()** never returns NULL. If the allocation fails, **pfMalloc()** generates a **pfNotify()** of level PFNFY_FATAL, so unless the application has set a **pfNotifyHandler()**, the application will exit.

Memory allocated with **pfMalloc()** must be freed with **pfFree()**, not with the standard C library's **free()** function. Using **free()** with data allocated by **pfMalloc()** will have devastating results.

Memory allocated with **pfMalloc()** has a reference count (see"pfDelete() and Reference Counting" in Chapter 2 for information on reference counting).

For example, if you use **pfMalloc()** to create attribute and index arrays, which you then attach to pfGeoSets using **pfGSetAttr()**, IRIS Performer automatically tracks the reference counts for the arrays, letting you delete the arrays much more easily than if you create them without **pfMalloc()**. All the reference-counting routines (including **pfDelete()**) work with data allocated using **pfMalloc()**. Note, however, that **pfFree()** doesn't check the reference count before freeing memory; use **pfFree()** only when you're sure the data you're freeing isn't referenced.

pfGetSize() returns the size in bytes of any data allocated by **pfMalloc()**. Since the size of such data is known, **pfCopy()** also works on allocated data.

Although **pfMalloc()**-allocated data behaves in many ways like a pfObject (see "Nodes" in Chapter 5), such data doesn't contain a user data pointer. This omission avoids requiring an extra word to be allocated with every piece of **pfMalloc()** data.

Note: All *libpr* objects are allocated using **pfMalloc()**, so you can use **pfGetArena()**, **pfGetSize()**, and **pfFree()** on all such objects. However, it's recommended that you use **pfDelete()** instead of **pfFree()** for *libpr* objects, in order to maintain reference-count checking.

Shared Arenas

pfInitArenas() creates two arenas, one for the allocation of shared memory with **pfMalloc()** and one for the allocation of semaphores and locks with **usnewlock()** and **usnewsema()**. The arenas are visible to all processes forked after **pfInitArenas()** is called.

Applications using *libpf* don't need to explicitly call **pfInitArenas()**, since it's invoked by **pfInit()**.

The shared memory arena can be allocated by memory-mapping either swap space (*/dev/zero*, the default) or an actual disk file (in the directory specified by the environment variable PFTMPDIR). The latter requires sufficient disk space for as much of the shared memory arena as will be used, and disk files are somewhat slower than swap space in allocating memory.

By default, IRIS Performer creates a large shared memory arena of 256 MB. Though this approach makes large numbers appear when you run *ps*(1), it

doesn't consume any substantial resources, since swap or file system space isn't actually allocated until accessed (that is, until **pfMalloc()** is called).

Because IRIS Performer cannot increase the size of the arena after initialization, an application requiring a larger shared memory arena should call **pfSharedArenaSize()** to specify the maximum amount of memory to be used. Arena sizes as large as 1.7 GB are usually acceptable; but you may need to set the virtual-memory-use and memory-use limits, using the shell *limit* command or **setrlimit()**, to allow your application to use that much memory.

Allocating Locks and Semaphores

An application requiring lockable pieces of memory should consider using pfDataPools, described below. Alternatively, when a lock or semaphore is required in an application that has called **pfInitArenas()**, you can call **pfGetSemaArena()** to get an arena pointer, and you can allocate locks or semaphores using **usnewlock()** and **usnewsema()**.

Datapools

Datapools, or pfDataPools, are also a form of shared memory, but they work differently from **pfMalloc()**. Datapools allow unrelated processes to share memory and lock out access to eliminate data contention. They also provide a way for one process to access memory allocated by another process.

Any process can create a datapool by calling **pfCreateDPool()** with a name and byte size for the pool. If an unrelated process needs access to the datapool, it must first put the datapool in its address space by calling **pfAttachDPool()** with the name of the datapool. The datapool must reside at the same virtual address in all processes. If the default choice of an address causes a conflict in an attaching process **pfAttachDPool()** will fail. To avoid this **pfDPoolAttachAddr()** can be called before **pfCreateDPool()** to specify a different address for the datapool.

Any attached process can allocate memory from the datapool by calling **pfDPoolAlloc()**. Each block of memory allocated from a datapool is assigned an ID so that other processes can retrieve the address using **pfDPoolFind()**.

Once you've allocated memory from a datapool, you can lock the memory chunk (not the entire pfDataPool) by calling **pfDPoolLock()** before accessing the memory. This locking mechanism works only if all processes wishing to access the datapool memory use **pfDPoolLock()** and **pfDPoolUnlock()**. After a piece of memory has been locked using **pfDPoolLock()**, any subsequent **pfDPoolLock()** call on the same piece of memory will block until the next time a **pfDPoolUnlock()** is called for that memory.

pfDataPools are pfObjects, so call **pfDelete()** to delete them. Calling **pfReleaseDPool()** unlinks the file used for the datapool—it doesn't immediately free up the memory that was used or prevent further allocations from the datapool; it just prevents processes from attaching to it. The memory is freed when the last process referring to the datapool **pfDelete()**s it.

CycleBuffers

A multiprocessed environment often requires that data be duplicated so that each process can work on its own copy of the data without adversely colliding with other processes. pfCycleBuffer is a memory structure which supports this programming paradigm. A pfCycleBuffer consists of one or more pfCycleMemories which are equally-sized memory blocks. The number of pfCycleMemories per pfCycleBuffer is global and is set once with **pfCBufferConfig()**, and is typically equal to the number of processes accessing the data.

Each process has a global index, set with **pfCurCBufferIndex()**, which indexes a pfCycleBuffer's array of pfCycleMemories. When each process has a different index (and its own address space), mutual exclusion is ensured if the process limits its pfCycleMemory access to the currently indexed one.

The "cycle" term of pfCycleBuffer refers to its suitability for pipelined multiprocessing environments where processes are arranged in stages like an assembly line and data propagates down one stage of the pipeline each frame. In this situation, the array of pfCycleMemories can be visualized as a circular list. Each stage in the pipeline accesses a different pfCycleMemory and at frame boundaries the global index in each process is advanced to the next pfCycleMemory in the chain. In this way, data changes made in the head of the pipeline are propagated through the pipeline stages by "cycling" the pfCycleMemories.

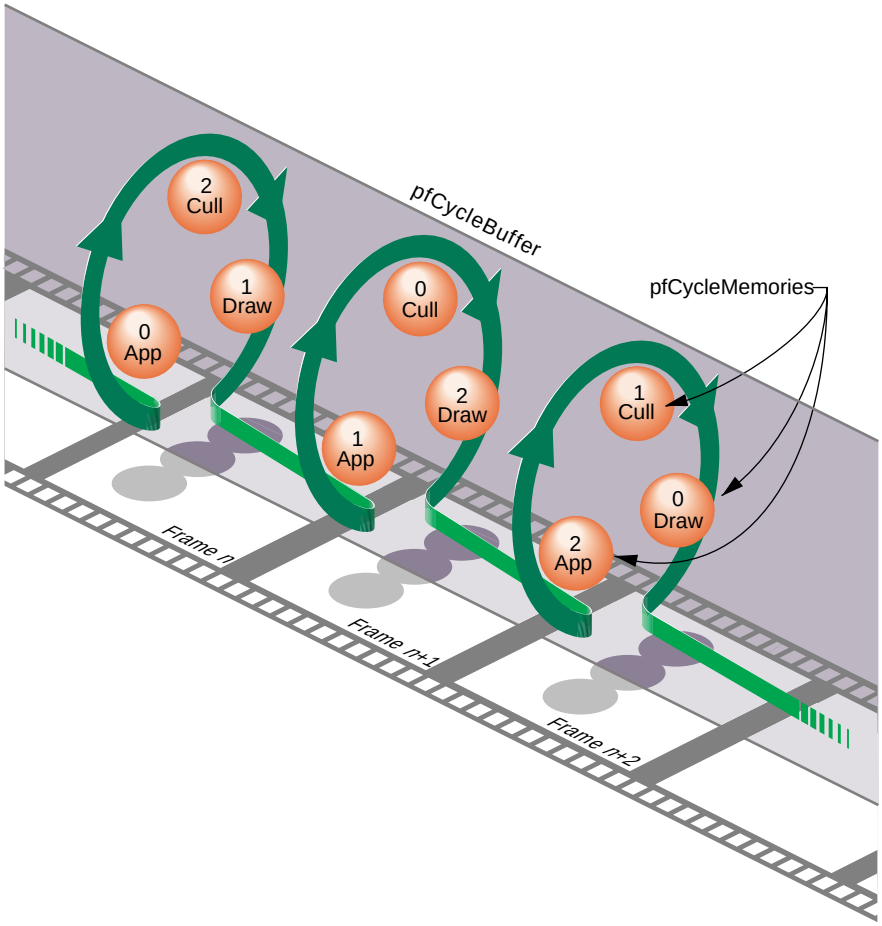


Figure 10-4 pfCycleBuffer and pfCycleMemory Overview

Cycling the memory buffers works if each current pfCycleMemory is completely updated each frame. If this is not the case, buffer cycling will eventually access a “stale” pfCycleMemory whose contents were valid some number of frames ago but are invalid now. pfCycleBuffers manage this by frame-stamping a pfCycleMemory whenever **pfCBufferChanged()** is called. The global frame count is advanced with **pfCBufferFrame()** which also copies most recent pfCycleMemories into “stale” pfCycleMemories thereby automatically keeping all pfCycleBuffers current.

A pfCycleBuffer consisting of pfCycleMemories of **nbytes** size is allocated from memory arena with **pfNewCBuffer(nbytes, arena)**. To initialize all the pfCycleMemories of a pfCycleBuffer to the same data call **pfInitCBuffer()**. pfCycleMemory is derived from pfMemory so you can use inherited routines like **pfCopy**, **pfGetSize**, and **pfGetArena()** on pfCycleMemories.

While pfCycleBuffers may be used for application data, their primary use is as pfGeoSet attribute arrays, e.g., coordinates or colors. pfGeoSets accept pfCycleBuffers (or pfCycleMemory) references as attribute references and automatically select the proper pfCycleMemory when drawing or intersecting with the pfGeoSet.

Note: *libpf* applications need not call **pfCBufferConfig()** or **pfCBufferFrame()** since the *libpf* routines **pfConfig()** and **pfFrame()** call these respectively.

Asynchronous I/O

A nonblocking file interface is provided to allow real-time programs access to disk files without affecting program timing. The system calls **pfOpenFile()**, **pfCloseFile()**, **pfReadFile()**, and **pfWriteFile()** work in an identical fashion to their IRIX counterparts **open()**, **close()**, **read()**, and **write()**.

When **pfOpenFile()** or **pfCreateFile()** is called, a new process is created using **sproc()**, which manages access to the file. Subsequent calls to **pfReadFile()**, **pfWriteFile()**, and **pfSeekFile()** place commands in a queue for the file manager to execute and return immediately. To determine the status of a file operation, call **pfGetFileStatus()**.

Error-Handling and Notification

IRIS Performer provides a general method for handling errors both within IRIS Performer and in the application. Applications can control error-handling by installing their own error-handling functions. You can also control the level of importance of an error.

Table 10-25 lists and describes the functions for setting notification levels.

Table 10-25 pfNotify Functions

Function	Action
pfNotifyHandler	Install user error-handling function.
pfNotifyLevel	Set the error-notification level.
pfNotify	Generate a notification.

pfNotify() allows an application to signal an error or print a message that can be selectively suppressed. **pfNotifyLevel()** sets the notification level to one of the values listed in Table 10-26.

Table 10-26 Error Notification Levels

Token	Meaning
PFNFY_ALWAYS	Always print regardless of notify level
PFNFY_FATAL	Fatal error
PFNFY_WARN	Serious warning
PFNFY_NOTICE	Warning
PFNFY_INFO	Information and floating point exceptions
PFNFY_DEBUG	Debug information
PFNFY_FP_DEBUG	Floating point debug information

The environment variable PFNFYLEVEL can be set to override the value specified in **pfNotifyLevel()**. Once the notification level is set via PFNFYLEVEL it can not be changed by an application.

Once the notify level is set, only those messages with a priority greater than or equal to the current level are printed or handed off to the user function. Fatal errors cause the program to exit unless the application has installed a handler with **pfNotifyHandler()**.

Setting the notification level to PFNFY_FP_DEBUG has the additional effect of trapping floating point exceptions such as overflows or operations on

invalid floating point numbers. It may be a good idea to use a notification level of PFNFY_FP_DEBUG while testing your application, so that you will be informed of all floating-point exceptions that occur.

File Search Paths

IRIS Performer provides a mechanism to allow referencing a file via a set of path names. Applications can create a search list of path names in two ways: the PFPATH environment variable and **pfFilePath()**. (Note that the PFPATH environment variable controls *file search paths* and has nothing to do with the pfPath data structure.)

Table 10-27 describes the routines for working with pfFilePaths.

Table 10-27 pfFilePath Routines

Routine	Action
pfFilePath	Create a search path.
pfFindFile	Search for the file using the search path.
pfGetFilePath	Supply current search path.

Pass a search path to **pfFilePath()** in the form of a colon-separated list of path names. Calling **pfFilePath()** a second time replaces the current path list rather than appending to it.

The environment variable PFPATH is also a colon-separated list of path names, similar to the PATH variable. **pfFindFile()** searches the paths in PFPATH first, then those given in the most recent **pfFilePath()** call; it returns the complete pathname for the file if the file is found. IRIS Performer applications should use **pfFindFile()** (either directly or through routines such as **pfdLoadFile()**) to look for input data files.

pfGetFilePath() returns the last search path specified by a **pfFilePath()** call. It doesn't return the path specified by the PFPATH environment variable—if you want to find out that value, call **getenv**(3c).

Chapter 11

“Math Routines”

This chapter details IRIS Performer’s comprehensive set of mathematical functions.

Chapter 11

Math Routines

This chapter describes the IRIS Performer math routines. Math routines let you create, modify, and manipulate vectors, matrices, line segments, planes, and bounding volumes such as spheres, boxes, and cylinders.

Vector Operations

A basic set of mathematical operations is provided for setting and manipulating floating point vectors of length 2, 3, and 4. The types of these vectors are pfVec2, pfVec3 and pfVec4, respectively. The components of a vector are denoted by PF_X, PF_Y, PF_Z, and PF_W with indices of 0, 1, 2, and 3, respectively. In the case of 4-vectors, the PF_W component acts as the homogeneous coordinate in transformations.

IRIS Performer supplies macro equivalents for many of the routines described in this section. Inlining the macros instead of calling the routines can substantially improve performance.

Table 11-1 lists the routines, what they do (in mathematical notation), and the macro equivalents (where available) for working with 3-vectors. Most of the same operations are also available for 2-vectors and 4-vectors, substituting “2” or “4” for “3” in the routine names. The only operations not available for 2-vectors are vector cross-products, point transforms, and vector transforms; the only operations unavailable for 4-vectors are vector cross-products and point transforms. (That is, there are no such routines as pfCrossVec2(), pfCrossVec4(), pfXformPt2(), pfXformPt4(), or pfXformVec2().)

Note: For the duration of this chapter, bold lowercase letters represent vectors and bold uppercase letters represent matrices. “X” indicates cross product, “·” denotes dot product, and vertical bars indicate the magnitude of a vector.

Table 11-1 Routines for 3-Vectors

Routine	Effect	Macro Equivalent
pfSetVec3(d, x, y, z)	$\mathbf{d} = (x, y, z)$	PFSET_VEC3
pfCopyVec3(d, v)	$\mathbf{d} = \mathbf{v}$	PFCOPY_VEC3
pfNegateVec3(d,v)	$\mathbf{d} = -\mathbf{v}$	PFNEGATE_VEC3
pfAddVec3(d, v1, v2)	$\mathbf{d} = \mathbf{v}_1 + \mathbf{v}_2$	PFADD_VEC3
pfSubVec3(d, v1, v2)	$\mathbf{d} = \mathbf{v}_1 - \mathbf{v}_2$	PFSUB_VEC3
pfScaleVec3(d, s, v)	$\mathbf{d} = s\mathbf{v}$	PFSCALE_VEC3
pfAddScaledVec3(d, v1, s, v2)	$\mathbf{d} = \mathbf{v}_1 + s\mathbf{v}_2$	PFADD_SCALED_VEC3
pfCombineVec3(d,s1,v1,s2,v2)	$\mathbf{d} = s_1\mathbf{v}_1 + s_2\mathbf{v}_2$	PFCOMBINE_VEC3
pfNormalizeVec3(d)	$\mathbf{d} = \mathbf{d} / \lvert\mathbf{d}\rvert$	none
pfCrossVec3(d, v1, v2)	$\mathbf{d} = \mathbf{v}_1 \times \mathbf{v}_2$	none
pfXformPt3(d, v, m)	$\mathbf{d} = \mathbf{vM}$, where $\mathbf{v} = (\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z,)$ and M is the 4x3 submatrix.	none
pfFullXformPt3(d, v, m)	$\mathbf{d} = \mathbf{vM} / \mathbf{d}_w$, where $\mathbf{v} = (\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z, 1)$	none
pfXformVec3(d, v, m)	$\mathbf{d} = \mathbf{vM}$, where $\mathbf{v} = (\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z, 0)$	none
pfDotVec3(v1, v2)	$\mathbf{v}_1 \cdot \mathbf{v}_2$	PFDOT_VEC3
pfLengthVec3(v)	$\lvert\mathbf{v}\rvert$	PFLENGTH_VEC3
pfSqrDistancePt3(v1, v2)	$\lvert\mathbf{v}_2 - \mathbf{v}_1\rvert^2$	PFSQR_DISTANCE_PT3
pfDistancePt3(v1, v2)	$\lvert\mathbf{v}_2 - \mathbf{v}_1\rvert$	PFDISTANCE_PT3

Table 11-1 (continued) Routines for 3-Vectors

Routine	Effect	Macro Equivalent
pfEqualVec3(v1, v2)	returns TRUE if $\mathbf{v}_1 = \mathbf{v}_2$, FALSE otherwise	PFEQUAL_VEC3
pfAlmostEqualVec3(v1,v2,tol)	returns TRUE if each element of $\mathbf{v}_1$ is within *tol* of the corresponding element of $\mathbf{v}_2$, FALSE otherwise	PFALMOST_EQUAL_VEC3

Matrix Operations

A pfMatrix is a 4x4 array of floating-point numbers that is used primarily to specify a transformation in homogeneous coordinates (*x*, *y*, *z*, *w*). Table 11-2 describes the IRIS Performer mathematical operations that act on matrices.

Table 11-2 Routines for 4x4 Matrices

Routine	Effect	Macro Equivalent
pfMakeIdentMat(d)	**D** = **I**	PFMAKE_IDENT_MAT
pfMakeVecRotVecMat(d,v1,v2)	**D** = **M** such that $\mathbf{v}_2 = \mathbf{v}_1\mathbf{M}$. $\mathbf{v}_1$, $\mathbf{v}_2$ normalized.	none
pfMakeQuatMat(d, q))	**D** = **M**, where **M** is the rotation of the quaternion *q*.	none
pfMakeRotMat(d, deg, x, y, z)	**D** = **M**, where **M** rotates by *deg* around (*x*, *y*, *z*)	none
pfMakeEulerMat(d, h, p, r)	**D** = **RPH**, where **R**, **P**, and **H** are the transforms for roll, pitch, and heading.	none

Table 11-2 (continued) Routines for 4x4 Matrices

Routine	Effect	Macro Equivalent
pfMakeTransMat(d, x, y, z)	**D** = **M**, where **M** translates by (*x*, *y*, *z*)	PFMAKE_TRANS_MAT
pfMakeScaleMat(d, x, y, z)	**D** = **M**, where **M** scales by (*x*, *y*, *z*)	PFMAKE_SCALE_MAT
pfMakeCoordMat(d, c)	**D** = **M**, where **M** rotates by (*h*, *p*, *r*) and translates by (*x*, *y*, *z*), with *h*, *p*, *r*, *x*, *y*, and *z* all specified by *c*	none
pfGetOrthoMatQuat(s, q)	returns, in q, a quaternion with the rotation specified by s.	none
pfGetOrthoMatCoord(s, d)	returns, in d, the rotation and translation specified by s	none
pfSetMatRow(d, r, x, y, z, w)	Set *r*th row of **D** equal to (x, y, z, w)	PFSET_MAT_ROW
pfGetMatRow(m, r, x, y, z, w)	(*x, *y, *z, *w) = *r*th row of **M**	PFGET_MAT_ROW
pfSetMatCol(d, c, x, y, z, w)	Set *c*th column of **D** equal to **(x, y, z, w)**	PFSET_MAT_COL
pfGetMatCol(m, c, x, y, z, w)	(*x, *y, *z, *w) = *c*th column of **M**	PFGET_MAT_COL
pfSetMatRowVec3(d, r, v)	Set *r*th row of **D** equal to **v**	PFSET_MAT_ROWVEC3
pfGetMatRowVec3(m, r, d)	**d** = *r*th row of **M**	PFGET_MAT_ROWVEC3
pfSetMatColVec3(d, c, v)	Set *c*th column of **D** equal to **v**	PFSET_MAT_COLVEC3
pfGetMatColVec3(m, c, d)	**d** = *c*th column of **M**	PFGET_MAT_COLVEC3

Table 11-2 (continued) Routines for 4x4 Matrices

Routine	Effect	Macro Equivalent
pfCopyMat(d, m)	**D** = **M**	PFCOPY_MAT
pfAddMat(d, m1, m2)	**D** = $\mathbf{M}_1$ + $\mathbf{M}_2$	none
pfSubMat(d, m1, m2)	**D** = $\mathbf{M}_1$ - $\mathbf{M}_2$	none
pfMultMat(d, m1, m2)	**D** = $\mathbf{M}_1\mathbf{M}_2$	none
pfPostMultMat(d, m)	**D** = **DM**	none
pfPreMultMat(d, m)	**D** = **MD**	none
pfTransposeMat(d, m)	**D** = **MT**	none
pfPreTransMat(d, m, x, y, z)	**D** = **TM**, where **T** translates by (*x*, *y*, *z*)	none
pfPostTransMat(d, x, y, z, m)	**D** = **MT**, where **T** translates by (*x*, *y*, *z*)	none
pfPreRotMat(d, deg, x, y, z, m)	**D** = **RM**, where **R** rotates by *deg* around (*x*, *y*, *z*)	none
pfPostRotMat(d, m, deg, x, y, z)	**D** = **MR**, where **R** rotates by *deg* around (*x*, *y*, *z*)	none
pfPreScaleMat(d, x, y, z, m)	**D** = **SM**, where **S** scales by (*x*, *y*, *z*)	none
pfPostScaleMat(d, m, x, y, z)	**D** = **MS**, where **S** scales by (*x*, *y*, *z*)	none
pfInvertFullMat(d, m))	**D** = $\mathbf{M}^{-1}$ for general matrices	none
pfInvertAffMat(d, m)	**D** = $\mathbf{M}^{-1}$, with **M** affine	none
pfInvertOrthoMat(d, m)	**D** = $\mathbf{M}^{-1}$, with **M** orthogonal	none
pfInvertOrthoNMat(d, m)	**D** = $\mathbf{M}^{-1}$, with **M** orthonormal	none

Table 11-2 (continued) Routines for 4x4 Matrices

Routine	Effect	Macro Equivalent
pfInvertIdentMat(d, m)	**D** = $\mathbf{M}^{-1}$, with **M** equal to the identity matrix	none
pfEqualMat(d, m)	returns TRUE if **D** = **M**, FALSE otherwise	PFEQUAL_MAT
pfAlmostEqualMat(d, m, tol)	returns TRUE if each element of **D** is within *tol* of the corresponding element of **M**, FALSE otherwise	PFALMOST_EQUAL_MAT

Some of the math routines that take a matrix as an argument are restricted to affine, orthogonal, or orthonormal matrices, these restrictions being noted by Aff, Ortho and OrthoN, respectively. (If such a restriction isn't noted in a *libpr* routine name, the routine can take a general matrix.)

An affine transformation is one that leaves the homogeneous coordinate unchanged—that is, in which the last column is (0,0,0,1). An orthogonal transformation is one that preserves angles. It can include translation, rotation, and uniform scaling, but no shearing or nonuniform scaling. An orthonormal transformation is an orthogonal transformation that preserves distances; that is, one that contains no scaling.

In the visual simulation library, *libpf*, most routines require the matrix to be orthogonal, although this isn't noted in the routine names.

The standard order of transformations for a hierarchical scene involves postmultiplying the transformation matrix for a child by the matrix for the parent. For instance, assume your scene involves a hand attached to an arm attached to a body. To get a transformation matrix H for the hand, postmultiply the arm's transformation matrix (A) by the body's (B): H = AB. To transform the hand object (at location h in hand coordinates) to body coordinates, calculate h' = hH.

Example 11-1 Matrix and Vector Math Examples

```
/*
 * test Rot of v1 onto v2
 */
{
    pfVec3 v1, v2, v3;
    pfMatrix m1;

    MakeRandomVec3(v1);
    MakeRandomVec3(v2);
    pfNormalizeVec3(v1);
    pfNormalizeVec3(v2);
    pfMakeVecRotVecMat(m1, v1, v2);
    pfXformVec3(v3, v1, m1);
    AssertEqVec3(v3, v2, “Arb Rot To”);
}

/*
 * test inversion of Affine Matrix
 */
{
    pfVec3 v1, v2, v3;
    pfMatrix m1, m2, m3;

    MakeRandomVec3(v3);
    pfMakeScaleMat(m2, v3[0], v3[1], v3[2]);
    pfPreMultMat(m1, m2);

    MakeRandomVec3(v1);
    pfNormalizeVec3(v1);
    MakeRandomVec3(v2);
    pfNormalizeVec3(v2);
    pfMakeVecRotVecMat(m1, v1, v2);
    s = pfLengthVec3(v2)/pfLengthVec3(v1);
    pfPreScaleMat(m1, s, s, s, m1);

    MakeRandomVec3(v1);
    pfNormalizeVec3(v1);
    MakeRandomVec3(v2);
    pfNormalizeVec3(v2);
    pfMakeVecRotVecMat(m2, v1, v2);

    MakeRandomVec3(v3);
    pfMakeTransMat(m2, v3[0], v3[1], v3[1]);
```

```
pfPreMultMat(m1, m2);

pfInvertAffMat(m3, m1);
pfPostMultMat(m3, m1);
AssertEqMat(m3, ident, “affine inverse”);
```

Quaternion Operations

A pfQuat is the IRIS Performer data structure (a pfVec4) whose for floating point values represent the components of a quaternion. Quaternions have many beneficial properties. The most relevant of these is their ability to represent 3D rotations in a manner that allows relatively simple yet meaningful interpolation between rotations. Much like multiplying two matrices, multiplying two quaternions results in the concatenation of the transformations. For more information on quaternions, see the article by Sir William Rowan Hamilton *“On quaternions; or on a new system of imaginaries in algebra,”* in the Philosophical Magazine, xxv, pp. 10-13 (July 1844), or refer to the sources noted in the pfQuat(3pf) reference page.

The properties of spherical linear interpolation makes quaternions much better suited than matrices for interpolating transformation values from keyframes in animations. The most common usage then is to use **pfSlerpQuat()** to interpolate between two quaternions representing two rotational transformations. The quaternion that results from the interpolation can then be passed to **pfMakeQuatMat()** to generate a matrix for use in a subsequent IRIS Performer call such as **pfDCSMat()**. While converting a quaternion to a matrix is relatively efficient, converting a matrix to a quaternion with **pfGetOrthoMatQuat()** is expensive and should be avoided when possible.

Because a pfQuat is also a pfVec4, all of the pfVec4 routines and macros may be used on pfQuats as well.

Table 11-3 Routines for Quaternions

Routine	Effect	Macro Equivalent
pfMakeRotQuat(q, a, x, y, z)	Sets q to rotation of a degrees about (x, y, z)	none
pfGetQuatRot(q, a, x, y, z)	Sets *a to angle and (*x, *y, *z) to axis of rotation represented by q	none
pfConjQuat(d, q)	d = conjugate of q	PFCONJ_QUAT
pfLengthQuat(q)	returns length of q	PFLENGTH_QUAT
pfMultQuat(d, q1, q2)	d = q1 * q2	PFMULT_QUAT
pfDivQuat(d, q1, d2)	d = q1 / q1	PFDIV_QUAT
pfInvertQuat(d, q1)	d = 1 / q1	
pfExpQuat(d, q)	d = exp(q)	none
pfLogQuat(d, q)	d = ln(q)	none
pfSlerpQuat(d, t, q1, q2)	d = interpolation with weight t between q1 (t=0.0) and q2 (t=1.0)	none
pfSquadQuat(d, t, q1, q2, a, b)	d = quadratic interpolation between q1 and q2	none
pfQuatMeanTangent(d, q1, q2, q3)	d = mean tangent of q1, q2 and q3.	none

Example 11-2 Quaternion Example

```
/*
 * test quaternion slerp
 */

    pfQuat q1, q2, q3;
    pfMatrix m1, m2, m3, m3q;
```

```
    pfVec3 axis;
    float angle1, angle2, angle, t;


    MakeRandomVec3(axis);
    pfNormalizeVec3(axis);
    angle1 = -drand48()*90.0f;
    angle2 = drand48()*90.0f;
    t = drand48();

    pfMakeRotMat(m1, angle1, axis[0], axis[1], axis[2]);
    pfMakeRotQuat(q1, angle1, axis[0], axis[1], axis[2]);
    pfMakeQuatMat(m3q, q1);

    pfMakeRotMat(m2, angle2, axis[0], axis[1], axis[2]);
    pfMakeRotQuat(q2, angle2, axis[0], axis[1], axis[2]);
    pfMakeQuatMat(m3q, q2);
    AssertEqMat(m2, m3q, “make rot quat q2”);

    angle = (1.0f-t) * angle1 + t * angle2;
    pfMakeRotMat(m3, angle, axis[0], axis[1], axis[2]);

    pfMakeRotQuat(q1, angle1, axis[0], axis[1], axis[2]);
    pfMakeRotQuat(q2, angle2, axis[0], axis[1], axis[2]);
    pfSlerpQuat(q3, t, q1, q2);
    pfMakeQuatMat(m3q, q3);
    AssertEqMat(m3q, m3, “quaternion slerp”);
{
```

Matrix Stack Operations

IRIS Performer allows you to create a stack of transformation matrices, which is called a pfMatStack.

Table 11-4 lists and describes the matrix stack routines. Note that none of these routines has a macro equivalent. The matrix at the top of the matrix stack is denoted “TOS,” for “Top of Stack.”

Table 11-4 Matrix Stack Routines

Routine	Operation
pfNewMStack	Allocate storage.
pfResetMStack	Reset the stack.
pfPushMStack	Duplicate the TOS and push it on the stack.
pfPopMStack	Pop the stack.
pfPreMultMStack	Premultiply the TOS by a matrix.
pfPostMultMStack	Postmultiply the TOS by a matrix.
pfLoadMStack	Set the TOS matrix.
pfGetMStack	Get the TOS matrix.
pfGetMStackTop	Get a pointer to the TOS matrix.
pfGetMStackDepth	Return the current depth of the stack.
pfPreTransMStack	Premultiply the TOS by a translation.
pfPostTransMStack	Postmultiply the TOS by a translation.
pfPreRotMStack	Premultiply thee TOS by a rotation.
pfPostRotMStack	Postmultiply the TOS by a rotation.
pfPreScaleMStack	Premultiply the TOS by a scale factor.
pfPostScaleMStack	Postmultiply the TOS by a scale factor.

Creating and Transforming Volumes

libpr provides a number of volume primitives, including sphere, box, cylinder, half-space (plane), and frustum. *libpf* uses the frustum primitive for a view frustum, and uses other volume primitives for bounding volumes:

- Nodes use bounding spheres.
- pfGeoSets use bounding boxes.
- Segments use bounding cylinders.

Defining a Volume

This section describes how to define geometric volumes.

Spheres

Spheres are defined by a center and a radius, as shown by the pfSphere structure's definition:

```
typedef struct {
    pfVec3 center;
    float radius;
} pfSphere;
```

A point **p** is in the sphere with center **c** and radius r if $|\mathbf{p} - \mathbf{c}| \leq r$.

Axially Aligned Boxes

An axially aligned box is defined by its two corners with the smallest and largest values for each coordinate. Its edges are parallel to the X, Y, and Z axes. It's represented by the pfBox data structure:

```
typedef struct {
    pfVec3 min;
    pfVec3 max;
} pfBox;
```

A point (x, y, z) is in the box if $min_x \leq x \leq max_x$, $min_y \leq y \leq max_y$, and $min_z \leq z \leq max_z$, .

Cylinders

A cylinder is defined by its center, radius, axis, and half-length, as shown by the definition of the pfCylinder data structure:

```
typedef struct {
    pfVec3 center;
    float radius;
    pfVec3 axis;
    float halfLength;
} pfCylinder;
```

A point **p** is in the cylinder with center **c**, radius r, axis **a**, and half-length h, if (**p** - **c**) • **a** $\leq$ h and | (1 - (**p** - **c**) • **a**) (**p** - **c**) | $\leq$ r.

Half-spaces (Planes)

A half-space is defined by a plane with a normal pointing away from the interior. It's represented by the pfPlane data structure:

```
typedef struct {
    pfVec3 normal;
    float offset;
} pfPlane;
```

A point **p** is in the half-space with normal **n** and offset d if **p** • **n** $\leq$ d.

Frusta

Unlike the other volumes, a pfFrustum isn't an exposed structure. You can allocate storage for a pfFrustum using **pfNewFrust()** and you can set the frustum using **pfMakePerspFrust()** or **pfMakeOrthoFrust()**.

Creating Bounding Volumes

The easiest and most efficient way to create a volume is to use one of the bounding operations. The routines in Table 11-5 create a bounding volume that encloses other geometric objects.

Table 11-5 Routines to Create Bounding Volumes

Routine	Bounding Volume
pfBoxAroundPts	Box enclosing a set of points
pfBoxAroundBoxes	Box enclosing a set of boxes
pfBoxAroundSpheres	Box enclosing a set of spheres
pfCylAroundSegs	Cylinder around a set of segments
pfSphereAroundPts	Sphere around a set of points
pfSphereAroundBoxes	Sphere around a set of boxes
pfSphereAroundSpheres	Sphere around a set of spheres

Bounding volumes can also be defined by extending existing volumes, but in many cases the tightness of the bounds created through a series of extend operations is substantially inferior to that of the bounds created with a single **pf*Around*()** operation.

Table 11-6 lists and describes the routines for extending bounding volumes.

Table 11-6 Routines to Extend Bounding Volumes

Routine	Operation
pfBoxExtendByPt	Extend a box to enclose a point.
pfBoxExtendByBox	Extend a box to enclose another box.
pfSphereExtendByPt	Extend a sphere to enclose a point.
pfSphereExtendBySphere	Extend a sphere to enclose a sphere.

Transforming Bounding Volumes

Transforming the volumes with an orthonormal transformation—that is, with no skew or nonuniform scaling, is straightforward for all of the volumes except for the axially aligned box. A straight transformation of the vertices doesn't suffice because the new box would no longer be axially aligned, so an aligned box must be created that encloses the transformed vertices. Hence a transformation of a box isn't generally reversed by applying the inverse transformation to the new box.

Table 11-7 lists and describes the routines that transform bounding volumes.

Table 11-7 Routines to Transform Bounding Volumes

Routine	Operation
pfOrthoXformPlane	Transform a plane or half-space.
pfOrthoXformFrust	Transform a frustum.
pfXformBox	Transform and extend a bounding box.
pfOrthoXformCyl	Transform a cylinder.
pfOrthoXformSphere r	Transform a sphere.

Intersecting Volumes

IRIS Performer provides a number of routines that test for intersection with volumes.

Point-Volume Intersection Tests

The point-volume intersection test returns PFIS_TRUE if the specified point is in the volume and PFIS_FALSE otherwise. Table 11-8 lists and describes the routines that test a point for inclusion within a bounding volume.

Table 11-8 Testing Points for Inclusion in a Bounding Volume

Routine	Test
pfBoxContainsPt	Point inside a box
pfSphereContainsPt	Point inside a sphere
pfCylContainsPt	Point inside a cylinder
pfHalfSpaceContainsPt	Point inside a half-space
pfFrustContainsPt	Point inside a frustum

Volume-Volume Intersection Tests

IRIS Performer provides a number of volume-volume tests that are used internally for bounding-volume tests when culling to a view frustum or when testing a group of line segments against geometry in a scene (see “Intersecting With pfGeoSets” on page 416). You can intersect spheres, boxes, and cylinders against half-spaces and against frustums for culling. You can intersect cylinders against spheres for testing grouped segments against bounding volumes in a scene.

Table 11-9 lists and describes the routines that test for volume intersections.

Table 11-9 Testing Volume Intersections

Routine	Action: Test if A Inside B
pfHalfSpaceContainsSphere	Sphere inside a half-space
pfFrustContainsSphere	Sphere inside a frustum
pfSphereContainsSphere	Sphere inside a sphere
pfSphereContainsCyl	Cylinder inside a sphere
pfHalfSpaceContainsCyl	Cylinder inside a half-space
pfFrustContainsCyl	Cylinder inside a frustum
pfHalfSpaceContainsBox	Box inside a half-space
pfFrustContainsBox	Box inside a frustum
pfBoxContainsBox	Box inside a box

The volume-volume intersection tests are designed to quickly locate empty intersections for rejection during a cull. If the complete intersection test is too time-consuming, the result PFIS_MAYBE is returned, to indicate that the two volumes might intersect.

The returned value is a bitwise OR of tokens, as shown in Table 11-10.

Table 11-10 Intersection Results

Test Result	Meaning
PFIS_FALSE	No intersection
PFIS_MAYBE	Possible intersection
PFIS_MAYBE | PFIS_TRUE	A contains at least part of B
PFIS_MAYBE | PFIS_TRUE | PFIS_ALL_IN	A contains all of B

This arrangement allows simple code such as that shown in Example 11-3.

Example 11-3 Quick Sphere Culling Against a Set of Half-Spaces

```
long
HSSContainsSphere(pfPlane **hs, pfSphere *sph, long numHS)
{
    long i, isect;

    isect = ~0;

    for (i = 0 ; i < numHS ; i++)
    {
        isect &= pfHalfSpaceContainsSphere(sph,hs[i]);
        if (isect == PFIS_FALSE)
            return isect;
     }
     /* if not ALL_IN all half spaces, don't know for sure */
     if (!(isect & PFIS_ALL_IN))
         isect &= ~PFIS_TRUE;
     return isect;
}
```

Creating and Working With Line Segments

A pfSeg represents a line segment starting at position *pos*, extending for a distance *length* in the direction *dir*:

```
typedef struct {
    pfVec3 pos;
    pfVec3 dir;
    float length;
} pfSeg;
```

The routines that operate on pfSegs assume that *dir* is of unit length and that *length* is positive; otherwise, the results of operations are undefined.

You can create line segments in three different ways:

- Specify a point and a direction directly in the structure—**pfSeg()**.
- Specify two endpoints: **pfMakePtsSeg()**.

- Specify one endpoint and an orientation in polar coordinates—**pfMakePolarSeg()**.
- Specify starting and ending distances along an existing segment—**pfClipSeg()**.

Intersection tests are the most important operations that use line segments. You can test the intersection of segments with volumes (half-spaces, spheres, and boxes), with 2D geometry (planes and triangles), and with geometry inside pfGeoSets.

Intersecting With Volumes

IRIS Performer supports intersections of segments with three types of convex volumes. **pfHalfSpaceIsectSeg()** intersects a segment with the half-space defined by a plane with an outward facing normal. **pfSphereIsectSeg()** intersects with a sphere and **pfBoxIsectSeg()** intersects with an axially aligned box.

The intersection test of a segment and a convex volume can have one of five results:

- The segment lies entirely outside the volume.
- The segment lies entirely within the volume.
- The segment lies partially inside the volume with the starting point inside.
- The segment lies partially inside the volume with the ending point inside.
- The segment lies partially inside the volume with both endpoints outside.

As with the volume-volume tests, the segment-volume intersection routines return a value that is the bitwise OR of some combination of the tokens PFIS_TRUE, PFIS_ALL_IN, PFIS_START_IN, and PFIS_MAYBE. (When PFIS_TRUE is set PFIS_MAYBE is also set for consistency with those routines that do quick intersection tests for culling.)

The functions take two arguments that return the distances along the segment of the starting and ending points. The return values are designed so

that you can AND them together for testing for the intersection of a segment against the intersection of a number of volumes. For example, a convex polyhedron is defined as the intersection of a set of half-spaces. Example 11-4 shows how to intersect a segment with a polyhedron.

Example 11-4 Intersecting a Segment With a Convex Polyhedron

```
long
HSSIsectSeg(pfPlane **hs, pfSeg *seg, long nhs, float *d1,
             float *d2)
{
    long retval = 0xffff;
    for (long i = 0 ; i < nhs ; i++)
    {
        retval &= pfHalfSpaceIsectSeg(hs[i], seg, d1, d2);
        if (retval == 0)
            return 0;
        pfClipSeg(seg, *d1, *d2);
    }
    return retval;
}
```

Note that the**se routines** do not actually clip the segment. If you want the segment to be clipped to the interior of the volume, you must call **pfClipSeg()**, as in the example above.

Intersecting With Planes and Triangles

Intersections with planes and triangles are simpler than those with volumes. **pfPlaneIsectSeg()** and **pfTriIsectSeg()** return either PFIS_TRUE or PFIS_FALSE, depending on whether an intersection has occurred. The distance of the intersection along the segment is returned in one of the arguments.

Intersecting With pfGeoSets

You can intersect line segments with the drawable geometry that's within pfGeoSets by calling **pfGSetIsectSegs()**. The operation is very similar to that of **pfNodeIsectSegs()**, except that rather than operating on an entire scene graph, only the triangles within the pfGeoSet are "traversed."

pfGSetIsectSegs() takes a pfSegSet and tests to see whether any of the segments intersect the polygons inside the specified pfGeoSet. By default, information about the closest intersection along each segment is returned as a set of pfHit objects, one for each line segment in the request. Each pfHit object indicates the location of the intersection, the normal, and what element was hit. This element identification includes the index of the primitive within the pfGeoSet and the triangle index within the primitive (for tristrips and quads primitives), as well as the actual triangle vertices.

You can also extract information from a pfHit object using **pfQueryHit()** and **pfMQueryHit()**. (See "Intersection Requests: pfSegSets" and "Intersection Return Data: pfHit Objects" in Chapter 6 for more information about pfSegSets and pfHit objects.) The principal difference between those routines and **pfGSetIsectSegs()** is that with **pfGSetIsectSegs()** information concerning the *libpf* scene graph (such as transformation, geode, name, and path) is never used.

Two types of intersection testing are possible, as shown in Table 11-11.

Table 11-11 Available Intersection Tests

Test Name	Function
PFTRAV_IS_GSET	Intersect the segment with the bounding box of the pfGeoSet.
PFTRAV_IS_PRIM	Intersect the segment with the polygon-based primitives inside the pfGeoSet.

You can use PFTRAV_IS_GSET for crude collision detection and PFTRAV_IS_PRIM for fine-grained testing. You can enable both bits and dynamically choose whether to go down to the primitive level by using a discriminator callback (see "Discriminator Callbacks"). **pfGSetIsectSegs()** performs only primitive-level testing for pfGeoSets consisting of triangles (PFGS_TRIS), quads (PFGS_QUADS), and tristrips (PFGS_TRISTRIPS), and all are decomposed into triangles.

Intersection Masks

Each pfGeoSet has an intersection mask that you set using **pfGSetIsectMask()**. The mask in the pfGeoSet is useful when pfGeoSets are embedded in a larger data structure; it allows you to define pfGeoSets to

belong to different classes of geometry for intersection—for example, water, ground, foliage. **pfGSetIsectSegs()** also takes a mask, and an intersection test is performed only if the bitwise AND of the two masks is nonzero.

Discriminator Callbacks

If a callback is specified in **pfGSetIsectSegs()**, that function is invoked when a successful intersection occurs, either with the bounding box of the pfGeoSet or with a primitive. The discriminator can decide what action to take based on the information about the intersection contained in a pfHit object. The return value from the discriminator determines whether the current intersection is valid and should be copied into the return structure, whether the rest of the geometry in the pfGeoSet is examined, and whether the segment should be clipped before continuing.

Unless the return value includes the bit PFTRAV_IS_IGNORE, the intersection is considered successful and is copied into the array of pfHit structures for return.

The bits of the PFTRAV_* tokens determine whether to continue, as shown in Table 11-12.

Table 11-12 Discriminator Return Values

Result	Meaning
PFTRAV_CONT	Continue examining geometry inside the pfGeoSet.
PFTRAV_PRUNE	Terminate the traversal now.
PFTRAV_TERM	Terminate the traversal now.

The bits PFTRAV_IS_CLIP_END and PFTRAV_IS_CLIP_START cause the segment to be clipped at the end or at the start using the intersection point. By default, in the absence of a discriminator, segments are end-clipped at each successful intersection at the finest level (bounding box or primitive level) requested. Hence, the closest intersection point is always returned.

The discriminator is passed a pfHit. You can use pfQueryHit() to examine information about the intersection, including which segment number within the pfSegSet the intersection is for and the current segment as clipped by previous intersections.

General Math Routine Example Program

Example 11-5 demonstrates the use of many of the available IRIS Performer math routines.

Example 11-5 Intersection Routines in Action

```
/*
 * simple test of pfCylIsectSeg
 */
{
    pfVec3 tmpvec;
    pfSetVec3(pt1, -2.0f, 0.0f, 0.0f);
    pfSetVec3(pt2, 2.0f, 0.0f, 0.0f);

    pfMakePtsSeg(&seg1, pt1, pt2);

    pfSetVec3(cyl1.axis, 1.0f, 0.0f, 0.0f);
    pfSetVec3(cyl1.center, 0.0f, 0.0f, 0.0f);
    cyl1.radius = 0.5f;
    cyl1.halfLength = 1.0f;

    isect = pfCylIsectSeg(&cyl1, &seg1, &t1, &t2);

    pfClipSeg(&clipSeg, &seg1, t1, t2);
    AssertFloatEq(clipSeg.length, 2.0f, “clipSeg.length”);
    pfSetVec3(tmpvec, 1.0f, 0.0f, 0.0f);
    AssertVec3Eq(clipSeg.dir, tmpvec, “clipSeg.dir”);
    pfSetVec3(tmpvec, -1.0f, 0.0f, 0.0f);
    AssertVec3Eq(clipSeg.pos, tmpvec, “clipSeg.pos”);
}
/*
 * simple test of pfTriIsectSeg
 */
{
    pfVec3 tr1, tr2, tr3;
    pfSeg seg;
    float d = 0.0f;
    long i;

    for (i = 0 ; i < 30 ; i++)
    {
    float alpha = 2.0f * drand48() - 0.5f;
    float beta = 2.0f * drand48() - 0.5f;
    float lscale = 2.0f * drand48();
```

```
    float target;
    long shouldisect;

    MakeRandomVec3(tr1);
    MakeRandomVec3(tr2);
    MakeRandomVec3(tr3);
    MakeRandomVec3(pt1);
    pfCombineVec3(pt2, alpha, tr2, beta, tr3);
    pfCombineVec3(pt2, 1.0f, pt2, 1.0f - alpha - beta, tr1);

    pfMakePtsSeg(&seg, pt1, pt2);
    target = seg.length;
    seg.length = lscale * seg.length;

    isect = pfTriIsectSeg(tr1, tr2, tr3, &seg, &d);
    shouldisect = (alpha >= 0.0f &&
                   beta >= 0.0f &&
                   alpha + beta <= 1.0f &&
                   lscale >= 1.0f);
if (shouldisect)
    if (!isect)
        printf("ERROR: missed\n");
    else
        AssertFloatEq(d, target, "hit at wrong distance");
 else if (isect)
    printf("ERROR: hit\n");
}

/*
 * simple test of pfCylContainsPt
 */
{
    pfCylinder cyl;
    pfVec3 pt;
    pfVec3 perp;

    pfSetVec3(cyl.center, 1.0f, 10.0f, 5.0f);
    pfSetVec3(cyl.axis, 0.0f, 0.0f, 1.0f);
    pfSetVec3(perp, 1.0f, 0.0f, 0.0f);
    cyl.halfLength = 2.0f;
    cyl.radius = 0.5f;

    pfCopyVec3(pt, cyl.center);
    if (!pfCylContainsPt(&cyl, pt))
        printf("center of cylinder not in cylinder!!!!\n");
```

```
    pfAddScaledVec3(pt, cyl.center, 0.9f*cyl.halfLength,
                    cyl.axis);
    if (!pfCylContainsPt(&cyl, pt))
        printf("0.9*halfLength not in cylinder!!!!\n");

    pfAddScaledVec3(pt, cyl.center, -0.9f*cyl.halfLength,
                    cyl.axis);
    if (!pfCylContainsPt(&cyl, pt))
        printf("-0.9*halfLength not in cylinder!!!!\n");

    pfAddScaledVec3(pt, cyl.center, -0.9f*cyl.halfLength,
                    cyl.axis);
    pfAddScaledVec3(pt, pt, 0.9f*cyl.radius, perp);
    if (!pfCylContainsPt(&cyl, pt))
        printf(printf("-0.9*halfLength not in cylinder!!\n");

    pfAddScaledVec3(pt, cyl.center, 0.9f*cyl.halfLength,
                    cyl.axis);
    pfAddScaledVec3(pt, pt, -0.9f*cyl.radius, perp);
    if (!pfCylContainsPt(&cyl, pt))
        printf("-0.9*halfLength not in cylinder!!!!\n");

    pfAddScaledVec3(pt, cyl.center, 1.1f*cyl.halfLength,
                    cyl.axis);
    if (pfCylContainsPt(&cyl, pt))
        printf("1.1*halfLength in cylinder!!!!\n");

    pfAddScaledVec3(pt, cyl.center, -1.1f*cyl.halfLength,
                    cyl.axis);
    if (pfCylContainsPt(&cyl, pt))
        printf("-1.1*halfLength in cylinder!!!!\n");

    pfAddScaledVec3(pt, cyl.center, -0.9f*cyl.halfLength,
                    cyl.axis);
    pfAddScaledVec3(pt, pt, 1.1f*cyl.radius, perp);
    if (pfCylContainsPt(&cyl, pt))
          printf("1.1*radius in cylinder!!!!\n");

    pfAddScaledVec3(pt, cyl.center, 0.9f*cyl.halfLength,
                    cyl.axis);
    pfAddScaledVec3(pt, pt, -1.1f*cyl.radius, perp);
    if (pfCylContainsPt(&cyl, pt))
        printf("1.1*radius in cylinder!!!!\n");
}
```

Chapter 12

“Statistics”

This chapter discusses the various kinds of available statistics on the performance of your application.

Chapter 12

Statistics

This chapter describes the IRIS Performer profiling utilities. Statistics are available on nearly every aspect of IRIS Performer's operation and can be used to diagnose both functionality and performance problems, as well as for writing benchmarks and for load management. For more detailed information on interpreting statistics to tune the performance of your application, refer to Chapter 13, "Performance Tuning and Debugging."

To collect most IRIS Performer statistics, all you have to do is enable them; IRIS Performer then collects them automatically for you in pfStats and pfFrameStats data structures (for *libpr* and *libpf*, respectively). You can query the contents of these structures from your program, or write the data to files. A *libpf* application can also display the contents of a pfFrameStats structure in a channel by calling **pfDrawChanStats()** or **pfDrawFStats()**. The statistics drawn for a channel are the statistics accumulated in the channel's own pfFrameStats. Such a display isn't necessary for statistics collection. The pointer to the pfFrameStats structure for a channel can be gotten with **pfGetChanFStats()**. You can then control which statistics for the channel are being accumulated.

Most of the IRIS Performer demo programs display some subset of these statistics. This chapter first explains some of the complex graphical displays and then discusses how to display statistics from a *libpf*-based application. Subsequent sections explain how to access and manipulate statistics from within an application. Topics include enabling and disabling statistics classes, printing, querying, and copying statistics data, as well as some basic examples showing common uses of statistics. At the end of this chapter is a discussion of the different statistics classes for *libpr* and *libpf*, along with details of their use.

Interpreting Statistics Displays

Many types of statistics can be displayed in a channel. Most such displays consist simply of labeled numbers and are fairly self-explanatory; however, some of the displays, such as the stage timing graph, warrant further explanation.

IRIS Performer tracks the time spent in the application, cull, and draw stages of the rendering pipeline. The basic statistics display shows a timing graph for each stage of the past several frames, as well as showing the current frame rate and load information. This profiling diagram is useful for optimizing both the database and application structure.

Figure 12-1 shows a sample stage timing graph from an IRIS Performer demo program. It might be helpful to refer to a running example as well—by turning on a statistics display in *perfly*, for instance—while reading this section.

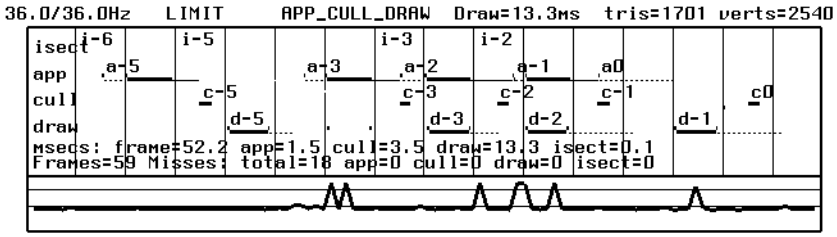


Figure 12-1 Stage Timing Statistics Display

The statistics diagram in Figure 12-1 is the simplest of the standard statistics displays. There are several other standard display formats, each emphasizing other classes of statistics. Statistics collection, though highly optimized, can take extra time in IRIS Performer operations. Because of this, you have a great deal of fine control over exactly what statistics are currently being collected and what statistics are being displayed. Statistics are divided into *classes* (separated into vertically stacked boxes in a display), and into *modes* within each class. The next several sections describe the classes shown in a typical statistics display.

Status Line

The top line of a standard statistics display, above the box that the rest of the statistics are drawn in, shows the current average frame rate followed by a slash and the target frame rate. (To set a target frame rate, call **pfFrameRate()**.) The rest of that status line indicates what frame-rate control method you're using (FLOAT, FREE, LIMIT, or LOCK—for details, see "Achieving the Frame Rate" in Chapter 7), your multiprocess model (set with **pfMultiprocess()**), and the average time (in milliseconds) spent in the channel draw callback. An optional part of the status line indicates the number of triangles in the scene.

Stage Timing Graph

The main part of the timing display is the stage timing graph, occupying the top portion of the statistics display. The red vertical lines (the darker ones in Figure 12-1) mark video retrace intervals, which occur at the video refresh rate of the system (commonly 60 times per second); a *field* is the period of time between two video retrace boundaries. The green vertical lines (the lighter ones in the figure) indicate frame boundaries. Note that frame boundaries are always on field boundaries and are an integral number of fields.

The segmented horizontal line segments in the top portion of the timing graph show the time taken by each of the IRIS Performer pipeline stages: i (intersection), a (application), c (cull), and d (draw) for each of the four frames shown (0 through 3). On screen, all stages belonging to a given frame are drawn in one color; different colors indicate different frames. The stages of the most recent frame, at the right of the graph, are marked a0, c0, and d0; previous frames have higher numbers (so "a-1" indicates the application stage of the immediately previous frame).

All stages performed by the same IRIX process are connected by vertical lines. If two stages are performed by different processes, they are not connected by a vertical line. In most multiprocessing modes, a stage of one frame occurs at the same time as another stage for a different frame, so that (for instance) d0 is directly below c-1 in the graph. The exception is the PFMP_CULLoDRAW model, in which the cull and draw stages for a given frame are performed in tandem; in this mode d0 is directly below c0 in the

graph. (In Figure 12-1, the PFMP_APP_CULL_DRAW model was used and all stages are part of separate processes.)

These stage timings are helpful when choosing a process model and balancing the cull and draw tasks for a database. Furthermore, the timing graph can show you how close you are to an improvement in frame rate as you view the database.

The timing lines for each stage are broken into pieces displayed at slightly different heights and thicknesses to show the time taken by significant subtasks within each stage. Raised segments reflect time spent in user code, intermediate lines reflect time spent in IRIS Performer code, and lowered lines reflect time waiting on other operations.

The application stage is divided into five subsegments:

- The time spent in the application's main loop between the **pfFrame()** call and the **pfSync()** call (drawn as a raised line).
- The time spent cleaning the scene graph during **pfSync()**.
- The time spent sleeping in **pfSync()** while waiting for the next frame boundary (drawn as a low thin pale dotted line).
- The time spent in the application code between calling **pfSync()** and calling **pfFrame()** (drawn as bright raised line).
- The time spent in **pfFrame()** cleaning the scene graph after any changes that might have been made in the previous subsegment, and then checking intersections.
- The time spent waiting while the cull process copies updated information from the application and starts the cull stage for the now-finished frame (drawn as a low thin line).

The cull stage is divided into only two subsegments:

- The time spent receiving updates from the application (in some multiprocessing models, this overlaps with the last subsegment of the application stage).
- The time spent in the channel cull callback, including time spent in **pfCull()** (drawn as raised line).

The draw stage has four subsegments:

- The time spent in the channel draw callback before the call to **pfDraw()** (a very short thick dark raised segment. This will include the time for your call to **pfClearChan()**. However, under normal circumstances, this segment shouldn't be visible at all). Operations taking place during this time should only be latency-critical. This line should be extremely short unless you have a very good reason.
- The time spent in **pfDraw()** (the main thick segment).
- The time spent in the channel draw callback after **pfDraw()** (another short thick dark raised segment;). Additionally, the last channel drawn on the pipe will include the time for the graphics pipeline to finish its drawing. Even if you have no operations after pfDraw() in you draw callback, this line for the last channel might look quite long, particularly if you are very fill-limited. It is possible for rendering calls issued in the previous section to fill up the graphics FIFO and have calls issued on this section have to wait while the graphics pipeline processes the commands and FIFO drains, making the time look longer than expected.
- The time spent waiting for the graphics pipeline to finish drawing the current frame, draw the channel statistics (for all channels), and swap color buffers. This line may look quite long as the color buffers can only be swapped on a vertical retrace boundary. Thus, this pale dotted line will always reach to the start of the next field.

Below the stage timing lines, the average time for each stage (in milliseconds) is shown. Note that the time given for the draw stage is the same as the time shown for the draw stage on the status line above the statistics box.

Load and Stress

The lower portion of the channel statistics diagram shows the recent history of graphics load and stress management. The load measure is based on the amount of time taken to draw previous frames in the channel relative to the specified goal frame time. A wavy red horizontal line is drawn to show the last three seconds of graphics load. A pair of white horizontal lines represent the upper and lower bounds of graphics load for invoking stress management. Thus, when the red line wanders outside the boundaries set by the white lines, stress management is invoked.

Stress management causes scaling of LODs in the database to meet the target frame rate with maximum scene detail. The last three seconds of stress are shown in white while stress management is running. Thus, the channel statistics graph can be used to tune the upper and lower bounds of the hysteresis band for invoking stress management and for tuning LODs of objects in the database.

CPU Statistics

CPU statistics, illustrated (with some other statistics) in Figure 12-2, give you information on system usage and load. The numbers shown correspond exactly to numbers given by *osview*; they're updated every update period just like other statistics (see "Setting Update Rate" on page 442 for information on how to change the update rate). These numbers represent averages (per second) across all CPUs; thus, if one or more CPUs is busy with some other task, the CPU statistics shown may not accurately reflect IRIS Performer CPU use. Note that the top line of the CPU statistics panel shows the total number of frames during the last update period, and the total time elapsed during that period.

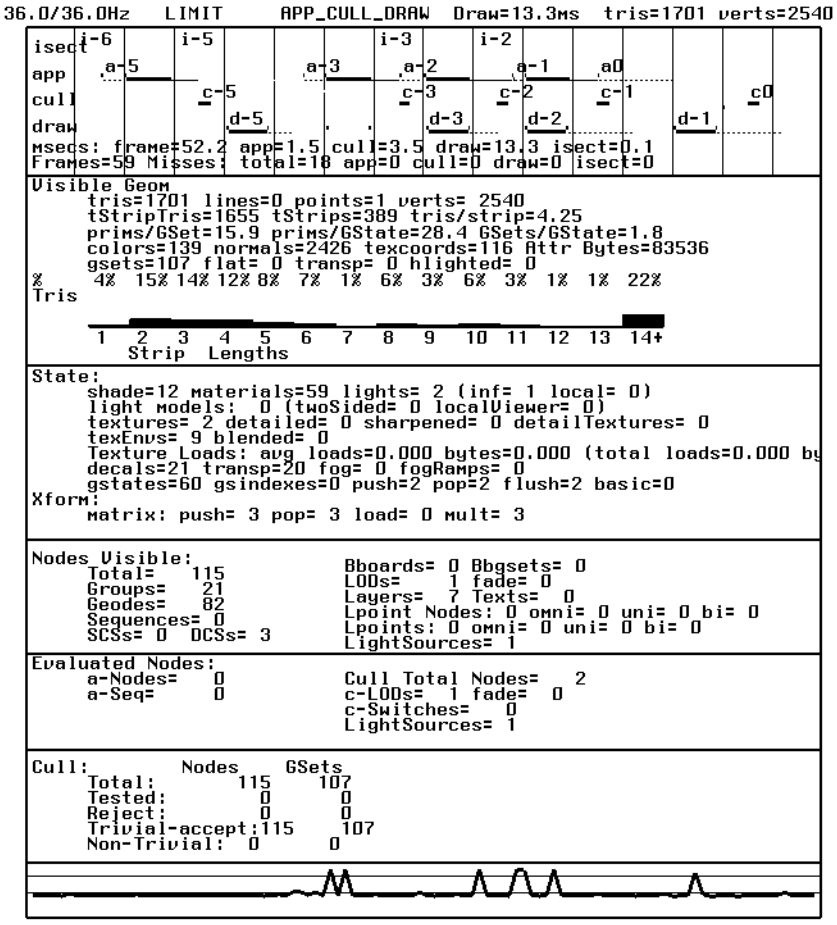


Figure 12-2 Other Statistics Classes

Rendering Statistics

Several other classes of statistics can be shown, each representing a different aspect of rendering performance. Some of these classes show:

- Data about visible geometry, including a histogram showing the percentage of triangles in the scene that are part of strips of a given length (from 1 to 14). Quads are counted as strips of length two; independent triangles count as strips of length one. This histogram is mostly useful as a diagnostic to see how well your database is structured for drawing efficiency; if it shows too many very short strips you may want to go back and restructure your database. (As a general rule of thumb, consider a "very short strip" to be one that's less than 4 triangles long but that number may vary depending on your database). To enable these statistics on a channel do:

  ```
  pfFStatsClass(pfGetChanFStats(chan),
      PFSTATS_ENGFX, PFSTATS_ON);
  ```

- A summary of the graphics state operations (including loading of textures), and of the number of operations that have recently been performed on the transformation stack (also part of the GFX stats class), the number of *libpf* nodes being drawn, in several categories (including billboards, light points, and geodes), plus the number of nodes of each type evaluated in the application and cull stages which can be enabled with

  ```
  pfFStatsClass(pfGetChanFStats(chan),
      PFSTATS_ENDB, PFSTATS_ON);
  ```

- Cull statistics, including how many nodes and pfGeoSets are being tested, how many are accepted, and how many are rejected by the *libpf* culling task, enabled with

  ```
  pfFStatsClass(pfGetChanFStats(chan),
      PFFSTATS_ENDB, PFSTATS_ON);
  ```

Fill Statistics

The fill statistics display indicates how many millions of pixels have been drawn since the last statistics update. (For information on setting the length of time between statistics updates, see "Setting Update Rate" on page 442).

It also computes the average *depth complexity* of the image, which is the average number of times each pixel was touched per frame.

The depth complexity of a scene is also be displayed in the main channel. Each pixel will be colored according to how many times that pixel was written to during display, rather than according to the current rendering modes. The colors used range from dark blue (not written to at all) to bright pink (written over many times). This color scheme is used in calculating fill statistics; the coloring is done whenever you gather fill statistics, even when you aren't displaying the totals in your channel statistics display.

Stencil planes are used to store the number of times a pixel is written and thus to calculate fill statistics. If *n* stencil planes are available, no more than 2^n writes to any given pixel will be counted. By default, the calculation of fill statistics uses three stencil planes; to change that default, call **pfStatsHwAttr()**.

Fill statistics are part of the *libpr* pfStats statistics but can be enabled on both pfStats and pfFrameStats classes. To enable fill statistics you simply do:

```
pfStatsClass(statsptr,
    PFSTATSHW_ENGFXPIPE_FILL, PFSTATS_ON);
```

To enable fill statistics for a channel's pfFrameStats do:

```
pfFStatsClass(pfGetChanFStats(chan),
    PFSTATSHW_ENGFXPIPE_FILL, PFSTATS_ON);
```

Examples of fill statistics can be found in perfly and in */usr/share/Performer/src/pguide/libpr/C/fillstats.c*.

Collecting and Accessing Statistics in Your Application

If you just want to bring up a statistics display in your application, you may not need to know details about the data structures used for statistics. If, however, you want to do more complicated statistics-handling (including collecting statistics without displaying them), you need more advanced information. This section provides a general overview of statistics manipulation, followed by subsections containing specific information.

If you use *libpf*, a wide variety of statistics-manipulation functions is available. If you use *libpr*, however, you must do some things on your own. For instance, you have to bind your own pfStats structure in which to accumulate statistics.

Furthermore, you can't access some kinds of statistics except through *libpf* calls—for instance, you can't get culling statistics using *libpr* calls. If you want full access to statistics, you must use *libpf*. There are, however, *libpr* routines that allow you to do your own cumulative totaling and averaging of collected statistics.

To create your own statistics display, enable the statistics classes you want to use and disable any modes you don't want to use. Then enable any relevant hardware, if necessary, with **pfEnableStatsHw()**.

To ensure the accuracy of timing with your rendering statistics, you want to flush the graphics pipeline before calling **pfGetTime()**. You can do this in IRIS GL with **finish()** and in OpenGL with **glFinish()**. These calls are expensive and shouldn't be done more than at the start and end of drawing in frame.

Displaying Statistics Simply

To put up a simple statistics display, all you have to do is call the function **pfDrawChanStats()** and pass it a pointer to the pfChannel whose statistics you want to display. The **pfDrawChanStats()** routine can be called from any process within the application; the statistics will be displayed in the channel specified.

If you want to display one channel's statistics in another channel, call **pfDrawFStats()**; for an example of this technique, as well as the enabling and disabling of every statistics class, see the statistics programming example in the file */usr/share/Performer/src/pguide/libpf/C/stats.c*.

By default, a statistics display shows all enabled statistics. If you want to show only a subset of the statistics you're collecting, call **pfChanStatsMode()** with an enabling bitmask indicating which classes are to be displayed.

Enabling and Disabling Statistics for a Channel

For efficiency, you may want to turn off statistics collection for a given channel when you're not displaying that channel's statistics. In particular, the stage timing statistics are enabled by default, so if you're using a channel whose statistics you don't care about, you should disable statistics for that channel. To turn off statistics for a channel, call:

```
pfFStatsClass(pfGetChanFStats(chan),
    PFSTATS_ALL, PFSTATS_OFF);
```

Use the same function with different parameters to enable all or specific classes of statistics for a channel. You can specify which classes to enable in order to minimize statistics collection overhead.

Statistics in *libpr* and *libpf*—pfStats Versus pfFrameStats

libpf statistics accumulate into a pfFrameStats structure to later be displayed, printed, queried, or otherwise operated on. The pfFrameStats structure actually contains four buffers of statistics: a buffer for the previous frame's statistics, a buffer of averaged statistics for the previous update period, a buffer of accumulated statistics for the current update period, and a buffer of statistics being accumulated for the current frame.

The pfFrameStats structure is built upon the *libpr* pfStats structure, so the pfFrameStats API includes routines to duplicate the functionality of pfStats. The duplicated API exists because the routines cannot be intermixed. pfStats routines can only be used on pfStats structures and pfFrameStats routines can only be called with pfFrameStats structures. However, pfstats classes and class modes (designated with the PFSTATS_ prefix) can be enabled on a pfFrameStats structure.

The pfStats statistics classes include the system and hardware statistics for the graphics pipeline and the CPU, as well the pixel fill statistics and rendering statistics on geometry, graphics state, and matrix transformations. Some of the *libpr* statistics commands, such as **pfEnableStatsHw(PFSTATSHW_ENGFXPIPE_FILL),** require an active graphics context and thus should only be called from the draw process. However, these commands are usually never necessary in a *libpf* application

because the pfFrameStats operation will handle these commands automatically.

Statistics Class Structures

The pfFrameStats structure and the pfStats structure are both inherited from the pfObject structure. Thus, you can use the pfObject routines (**pfCopy()**, **pfPrint()**, **pfDelete()**, **pfUserData()**, **pfGetType()**, and so on) with pfStats and pfFrameStats structures. However, some pfObject routines will not support all of the semantics of a pfStats or pfFrameStats structure, so some pfStats versions of a few of these routines take extra arguments. These routines will have a pfFrameStats version as well. In particular, **pfCopyStats()** and **pfCopyFStats()** should be used to copy pfStats and pfFrameStats structures, respectively.

Routines that have “FStats” in their names (rather than just “Stats”) expect to be passed a full pfFrameStats structure rather than a pfStats structure. The pfFrameStats API includes additional routines beyond pfStats for supporting libpf statistics. For example, **pfDrawFStats()** to display statistics in a channel and **pfFStatsCountNode()** to accumulate the static database statistics for the scene graph rooted at the provided node. Additionally, pfFrameStats has special support for the multiprocessed environment of libpf and ensures that the statistics operations are all done in the correct process. All modifying of a pfFrameStats structure, including enabling and disabling of classes, printing, and copying, should all be done in the application process. **pfDrawFStats()** and **pfDrawChanStats()** can be called in either the application process or the draw process.

Statistics Rules of Use

Enabling and disabling of statistics and setting of modes and attributes on a statistics structure should always be done in the application process; the settings will automatically be passed down the process pipeline. To enable classes of statistics on a pfFrameStats, call **pfFStatsClass()** and provide a statistics structure, a bitmask of statistics-enabling tokens (tokens with “STATS_EN” in their names) for the desired classes, and the token PFSTATS_ON. Obtain the statistics structure from the desired channel by calling **pfGetChanFStats()**. For example,

```
pfFStatsClass(pfGetChanFStats(chan), PFFSTATS_ENCULL |
    PFFSTATS_ENDB, PFSTATS_ON);
```

enables the cull statistics and database statistics classes, leaving settings alone for any other classes. Notice that the classes specific to pfFrameStats have a PFFSTATS_ prefix. You can use PFSTATS_SET instead of PFSTATS_ON to enable only the specified classes (disabling any others that might already be enabled).

Statistics Tokens

There are five main types of statistics tokens:

- Statistics class enable bitmasks, used for selecting a set of classes to enable with **pfStatsClass()**. These bitmasks are also used when printing with **pfPrint()** or copying with **pfCopy()** and **pfCopyStats()**, as well as with the **pfResetStats()**, **pfClearStats()**, **pfAverageStats()**, and **pfAccumulateStats()** routines (and their pfFrameStats counterparts). These tokens are of the form PFSTATS_EN* and PFFSTATS_EN* for pfStats and pfFrameStats class respectively. The PFSTATS_ALL token will select all statistics classes, and also all statistics buffers in the case of a pfFrameStats structure. The token PFSTATS_EN_MASK selects all pfStats classes and the token PFFSTATS_EN_MASK selects all pfFrameStats statistics classes, which includes all pfStats classes.
- Value tokens, used to specify how to set a value for a specified pfStats or pfFrameStats class enable or mode. Value tokens include PFSTATS_ON, PFSTATS_OFF, and PFSTATS_DEFAULT. Another value token, PFSTATS_SET, is used to specify that the entire class enable or mode bitmask should be set to the specified mask. These tokens are used in conjunction with the class bitmasks and the class name tokens for **pfStatsClass()** and **pfStatsClassMode()**.
- Class name tokens, used to name a specific class. For instance, these tokens can be passed to **pfStatsClassMode()** to set individual modes of a statistics class.
- Class mode tokens, of which each statistics class has its own, and which have the form PFSTATS_class_mode and PFFSTATS_class_mode for pfStats and pfFrameStats class modes, respectively.

- Statistics query tokens, used with **pfQueryStats()**, **pfMQueryStats()**, **pfQueryFStats()**, and **pfMQueryFStats()**. These tokens are of the form PFSTATSVAL_* and PFFSTATSVAL_* and have matching pfStatsVal* types for holding the returned data. The token PFFSTATS_BUF_MASK selects the pfFrameStats statistics buffers.

Statistics Buffers

You can only access the PREV and CUM statistics buffers from the IRIS Performer application process. Statistics from desired buffers in other processes should be queried in the application process and then passed down the process pipeline, which you can do using the channel data utility.

The AVG buffer is copied down the process pipeline at the end of each update period and so is available to by queried by other processes. The CUR statistics buffer is the current working area and contains the statistics accumulated so far from previous stages current frame; the contents of the CUR buffer is very dependent on the multiprocess configuration (but is almost always empty in the application process, so queries should access the PREV buffer). Statistics that are added to the CUR buffer via copying, accumulation, or immediate mode collection (such as with **pfStatsCountGSet()** and **pfFStatsCountNode()**) will be propagated down the process pipeline and then back up to the application process to be included in the PREV buffer.

In a *libpf* application, most statistics collection is completely automatic. The application must simply enable the desired classes of statistics via **pfStatsClass()** and / or **pfStatsClassMode()**.

The IRIS Performer processes are responsible for actually opening the pfFrameStats structure in which to accumulate the enabled statistics classes, as well as for managing any statistics hardware resources. All types of *libpf* statistics can be accumulated without ever making specific calls to open a structure for accumulation or enabling statistics hardware.

When using only *libpr* statistics, however, one must explicitly open a pfStats structure for statistics accumulation by calling **pfOpenStats()**.

Hardware statistics resources must also be managed by an application using only *libpr* statistics. Statistics function calls that have “HW” in their names, such as **pfEnableStatsHw()** and **pfStatsHwAttr()**, directly access system

hardware (such as graphics hardware and CPU); be careful to make such calls only from the relevant process. **pfEnableStatsHw()** expects PFSTATSHW_EN* bitmask tokens. Statistics classes which have corresponding statistics hardware have a PFSTATSHW_ prefix in their token names.

In a *libpf* application, IRIS Performer takes care of enabling the correct hardware modes that correspond to enabled classes of statistics. For more information about specific statistics classes, see the pfFrameStats(3pf) and pfStats(3pf) reference pages.

Reducing the Cost of Statistics

Collecting and displaying statistics can have a big impact on performance. This section describes ways to reduce that impact.

Enabling Only Statistics of Interest

Each channel has its Process Frame Times (PFTIMES) statistics class enabled by default. This class maintains a short history of process frames times, as well as averaging frame times over the default update period of two seconds.

To minimize unnecessary overhead, turn off statistics on channels when you're not using them. To turn off all statistics for a channel, call **pfFStatsClass()** in the application process with the statistics structure of the given channel:

```
pfFStatsClass(pfGetChanFStats(chan), PFSTATS_ALL,
    PFSTATS_OFF);
```

Each statistics class has default mode settings. The short history of process frame time is used to draw the timing graph. By default, this history consists of four frames of each IRIS Performer process (app, cull, draw, intersections).

Maintaining this short history of statistics can be disabled by calling **pfStatsClassMode()** with the token PFFSTATS_PFTIMES_HIST:

```
pfStatsClassMode(fstats, PFFSTATS_PFTIMES,
    PFFSTATS_PFTIMES_HIST, PFSTATS_OFF);
```

This is useful if you're only interested in the average frame times of each task with minimal overhead and you don't need to display the timing graph. However, for most applications, the overhead incurred by keeping the timing history is not noticeable.

Controlling Update Rate

The update rate controls how often statistics are averaged and new results are made available in the AVG buffer for display or query. Change the update rate by calling

```
pfFStatsAttr(fstats, {PFFSTATS_UPDATE_FRAMES,
    PFFSTATS_UPDATE_SECS}, val);
```

When the update rate is nonzero, statistics are accumulated every frame. When the update period is set to zero, no statistics accumulation or averaging is done and only statistics in the PREV and CUR buffers are maintained.

When statistics are accumulated and averaged, the averaging happens only in the application process, but accumulation is done in each IRIS Performer process.

Statistics Output

Once you've collected some statistics, you need to be able to access and manipulate them.

Printing

To print the contents of pfStats and pfFrameStats structures, use the general **pfPrint()** routine. The verbosity-level parameter to **pfPrint()** sets the level of detail to use in printing statistics. Statistics class-enable bitmasks can be used to select a subset of statistics to print. For instance, to print only the enabled statistics:

```
pfPrint(stats, pfGetStatsClass(stats, PFSTATS_ALL),
    PFPRINT_VB_INFO, 0);
```

When printing the contents of pfFrameStats structures, you can select which buffers are to be printed: PREV, CUR, AVG, or CUM. The selected statistics

from all selected buffers are printed. The following call prints the currently enabled statistics from the previous frame and from the averaged statistics buffer:

```
pfPrint(stats, PFFSTATS_BUF_PREV | PFFSTATS_BUF_AVG |
    pfGetStatsClass(stats, PFSTATS_ALL),
    PFPRINT_VB_INFO, 0);
```

Copying

You can copy entire pfStats and pfFrameStats structures with the general **pfCopy()** command. **pfCopy()** copies all of the statistics data, as well as information on mode settings and which classes are enabled. The source and destination structures must be of the same type. If both statistics structures are pfFrameStats structures, then all statistics from all buffers are copied.

The **pfCopyStats()** and **pfCopyFStats()** routines copies only statistics data (not class enables or mode settings) and accepts a class enable bitmask to select statistics classes for the copy. For example:

```
pfCopyStats(statsA, statsB, pfGetStatsClass(statsB,
    PFSTATS_ALL));
```

For a pfFrameStats structure, a PFFSTATS_BUF_* token can be included in the stats enable bitmask to select the buffer to be accessed. If no buffer is specified, the CUR buffer is used. For example,

```
pfCopyFStats(fstats, stats, PFFSTATS_BUF_PREV |
    pfGetStatsClass(stats, PFSTATS_ALL));
```

copies the currently enabled classes of *stats* to the PREV pfStats in *fstats*.

In this case, it's an error to select more than one statistics buffer, so PFSTATS_ALL cannot be used as the select. If you specify two pfFrameStats structures, the buffer select is used for both structures; if you select multiple buffers then each selected statistics class from each selected buffer is copied. The **pfCopyFStats()** routine allows you to copy between two different buffers of two pfFrameStats structures.

This routine takes explicit specification of PFFSTATS_BUF_* selects for source and destination. Any PFFSTATS_BUF_* included with the class enable bitmask is simply ignored, making it safe to specify PFSTATS_ALL. This routine will not accept a pfStats structure.

Querying

pfQueryStats() and **pfMQueryStats()** (and corresponding pfFrameStats versions) can be used to get values of out a pfStats or pfFrameStats structure and into an exposed structure.

These routines are useful when you want to use specific statistics for your own custom load management or for benchmarking, and you can use them to implement your own custom statistics utility routines. **pfQueryStats()** and **pfMQueryStats()** both take a pfStats (or pfFrameStats for **pfQueryFStats()** and **pfMQueryFStats()**) and return the number of bytes written to the provided destination buffer. **pfQueryStats()** takes a token that specifies a single query, while **pfMQueryStats()** expects a token buffer for multiple queries. If an error is encountered, both query routines immediately halt and return with the total number of bytes successfully written.

There are specific tokens for querying individual values or entire classes of statistics. The query tokens are of the form PFSTATSVAL_* and PFFSTATSVAL_*, and the corresponding exposed structure names are of the form pfStatsVal* and pfFStatsVal*. Queries on pfFrameStats structures with PFFSTATSVAL_* tokens expect a PFFSTATS_BUF_* select token to be ORed with the query select. It's an error to include more than one pfFrameStats buffer select token. If no buffer select token is provided, the CUR buffer will be queried. The statistics query tokens and structures are defined in *prstats.h* and *pfstats.h*.

Customizing Displays

The standard statistics displays have several parameters hard-wired in. For instance, you can't change the colors used in such displays. If you want to use different colors, you'll have to use your own display routines.

Setting Update Rate

To set the frequency at which statistics are automatically collected, use **pfFStatsAttr()**. See the **pfFrameStats**(3pf) reference page for details. If you want to turn off cumulative statistics collection (and thus running averages) entirely, set the update rate to zero. (Note that doing this will change your

statistics display; in particular, your actual frame rate will be changed to "???" and other averages will not be displayed.)

The pfStats Data Structure

The pfStats data structure contains four statistics buffers: one for current statistics, one for previous statistics, one for cumulative statistics, and one for averages.

If you're using *libpf* calls to have IRIS Performer keep track of statistics for you, you should always look at the previous-stats buffer; the current-stats buffer is kept in a state of flux, and if you look at it you're likely to find meaningless numbers there.

If, on the other hand, you're using *libpr* and keeping track of your own statistics, the current-stats buffer does contain accurate information.

Statistics Examples

This section contains some examples of statistics calls.

Setting statistics class enables and modes

Set all stats class enables on a pfStats to their default values:

```
pfStatsClass(stats, PFSTATS_ALL, PFSTATS_DEFAULT);
```

Set all modes for the PFSTATS_GFX class on a pfFrameStats to their default values:

```
pfFStatsClassMode(fstats, PFSTATS_GFX, PFSTATS_ALL,
    PFSTATS_DEFAULT);
```

Note that **pfStatsClassMode()** takes a class name as its class specifier (second argument) and not a bitmask. However, you can use PFSTATS_CLASS to refer to the modes of all classes;

```
pfFStatsClassMode(fstats, PFSTATS_MODE, PFSTATS_ALL,
    PFSTATS_DEFAULT);
```

sets all modes of all pfStats classes to their default values. For pfFrameStats classes, there is PFFSTATS_CLASS:

Set the entire class enable mask to all PFSTATS_ALL, effectively enabling all statistics classes:

```
pfFStatsClass(fstats, PFFSTATS_ALL, PFSTATS_SET);
```

Force off all modes of the PFSTATS_GFX class of a pfStats:

```
pfStatsClassMode(stats, PFSTATS_GFX, PFSTATS_OFF,
    PFSTATS_SET);
```

To track triangle strip lengths on a pfFrameStats, enable the graphics statistics class mode:

```
pfFStatsClassMode(fstats, PFSTATS_GFX,
    PFSTATS_GFX_TSTRIP_LENGTHS, PFSTATS_ON);
```

Chapter 13

“Performance Tuning and Debugging”

This chapter explains how to use performance measurement and debugging tools and provides hints for deriving maximum performance from your applications.

Chapter 13

Performance Tuning and Debugging

This chapter provides some basic guidelines to follow when tuning your application for optimal performance. It also describes how to use debugging tools like *pixie*, *prof*, *gldebug*, and *glprof* to debug and tune your applications. It concludes with some specific notes on tuning applications on systems with RealityEngine graphics.

Performance-Tuning Overview

This section contains some general performance-tuning principles. Some of these issues are discussed in more detail later in this chapter.

- Remember that high performance doesn't come by accident. You must design your programs with speed in mind for optimal results.
- Tuning graphical applications, particularly IRIS Performer applications, requires a pipeline-based approach. You can think of the graphics pipeline as comprising three separate stages; the pipeline runs only as fast as the slowest stage, so improving the performance of the pipeline requires improving the slowest stage's performance. The three stages are:
  - The host (or CPU) stage, in which routines are called and general processing is done by the application. This stage can be thought of as a software pipeline, sometimes called the *rendering pipeline*, itself comprising up to three sub-stages—the application, cull, and draw stages—as discussed at length throughout this guide.

  - The transformation stage, in which transformation matrices are applied to objects to position them in space (this includes matrix operations, setting graphics modes, transforming vertices, handling lighting, and clipping polygons).
  - The fill stage, which includes clearing the screen and then drawing and filling polygons (with operations such as Gouraud shading, z-buffering, and texture mapping).

- You can estimate your expected frame rate based on the makeup of the scene to be drawn and graphics speeds for the hardware you're using. Be sure to include fill rates, polygon transform rates, and time for mode changes, matrix operations, and screen clear in your calculations.
- Measure both the performance of complex scenes in your database and of individual objects and drawing primitive to verify your understanding of the performance requirements.
- Use the IRIS Performer diagnostic statistics to evaluate how long each stage takes and how much it does. See Chapter 12, "Statistics," for more information. These statistics are referred to frequently in this chapter.
- Use system tools to help profile and analyze your application. The IRIS GL *glprof* utility (described in "Using glprof to Find Performance Bottlenecks" on page 469) will profile the rendering of an IRIS GL programs and show what was drawn and help figure out what stage of the graphics hardware pipeline is the significant bottleneck.
- Tuning an application is an incremental process. As you improve one stage's performance, bottlenecks in other stages may become more noticeable. Also, don't be discouraged if you apply tuning techniques and find that your frame rate doesn't change—frame rates only change by a field at a time (which is to say in increments of 16.67 milliseconds for a 60 Hz display), while tuning may provide speed increases of finer granularity than that. To see performance improvements that don't actually increase frame rate, look at the times reported by IRIS Performer statistics on the cull and draw processes (see Chapter 12 for more information).
- See the graphics library books listed in the "Bibliography" on page xxix for information about how to get peak performance from your graphics hardware, beyond what IRIS Performer does for you.

How IRIS Performer Helps Performance

IRIS Performer uses many techniques to increase application performance. Knowing about what IRIS Performer is doing and how it affects the various pipeline stages may help you write more efficient code. This section lists some of the things IRIS Performer can do for you.

Draw Stage and Graphics Pipeline Optimizations

During drawing, IRIS Performer

- Sets up an internal record of what routines and rendering methods are fastest for the current graphics platform. This information can be queried in any process with **pfQueryFeature()**. You can use this information at run time when setting state properties on your pfGeoStates.
- Has machine-dependent fast options for commands that are very performance sensitive. Use the _ON and _FAST mode settings whenever possible to get machine-dependent high-performance defaults. Some examples include:
  - **pfAntialias(PFAA_ON)**
  - **pfTransparency(PFTR_ON)**
  - **pfDecal(PFDECAL_BASE_FAST)**
  - **pfTexFilter(tex, filt, PFTEX_FAST)**
  - **pfTevMode(tev, PFTEV_FAST)**
- Sets up default modes for drawing, multiprocessing, statistics, and other areas, that are chosen to provide high scene quality and performance. Some rendering defaults differ from GL defaults: backface elimination is enabled by default (**pfCullFace(PFCF_BACK)**) and lighting materials use **lmcolor()** in IRIS GL and **glColorMaterial()** in OpenGL to minimize the number of materials required in a database (**pfMtlColorMode(mtl, side, PFMTL_CMODE_AD)**).
- Uses a large number of specialized routines for rendering different kinds of objects extremely quickly. There's a specialized drawing routine for each possible pfGeoSet configuration (each choice of primitive type, attribute bindings, and index selection). Each time you

change from one pfGeoSet to another, one of these specialized routines is called. However, this happens even if the new pfGeoSet has the same configuration as the old one, so larger pfGeoSets are more efficient than smaller ones—the function-call overhead for drawing many small pfGeoSets can reduce performance. As a rule of thumb, a pfGeoSet should contain at least 4 triangles, and preferably between 8 and 64. If the pfGeoSet is too large, it can reduce the efficiency of other parts of the process pipeline.

- Caches state changes, because applying state changes is costly in the draw stage. IRIS Performer accumulates state changes until you call one of the **pfApply*()** functions, at which point it applies all the changes at once. Note that this differs from the graphics libraries, in which state changes are immediate. If you have several state changes to make in IRIS Performer, set up all the changes before applying them, rather than using the one-change-at-a-time approach (with each change followed by an apply operation) that you might expect if you're used to graphics library programming.
- Evaluates state changes lazily—that is, it avoids making any redundant changes. When you apply a state change, IRIS Performer compares the new graphics state to the previous one to see if they're different. If they are, it checks whether the new state sets any modes. If it does, IRIS Performer checks each mode being set to see whether it's different from the previous setting. To take advantage of this feature, share pfGeoStates and inherit states from the global state wherever possible. Set all the settings you can at the global level and let other nodes inherit those settings, rather than setting each node's attributes redundantly. To do this within a database, you can set up pfGeoStates with your desired global state and apply them to the pfChannel or pfScene with **pfChanGState()** or p**fSceneGState()**. You can do this through the database loader utilities in *libpfdu* trivially for a given scene with **pfdMakeSharedScene()**, or have more control over the process with **pfdDefaultGState()**, **pfdMakeSceneGState()**, and **pfdOptimizeGStateList()**.
- Provides an optimized immediate mode rendering display list data type, pfDispList, in *libpr*. The pfDispList type reduces host overhead in the drawing process and requires much less memory than a graphics library display list. *libpf* uses pfDispLists to pass results from the cull process to the draw process when the PFMP_CULL_DL_DRAW mode is turned on as part of the multiprocessing model. For more

information about display lists, see “Display Lists” in Chapter 10; for more information about multiprocessing, see “Successful Multiprocessing With IRIS Performer” in Chapter 7.

Cull and Intersection Optimizations

To help optimize culling and intersection, IRIS Performer

- Provides **pfFlatten()** to resolve instancing (via cloning) and static matrix transformations (by pre-transforming the cloned geometry). It can be especially important to flatten small pfGeoSets; otherwise matrix transformations must be applied to each small pfGeoSet at significant computational expense. Note that flattening resolves only static coordinate systems, not dynamic ones, but that where desired, pfDCS nodes can be converted to pfSCS nodes automatically using the IRIS Performer utility function **pfdFreezeTransforms()**, which allows for subsequent flattening. Using **pfFlatten()**, of course, increases performance at the cost of greater memory use. Further, the function **pfdCleanTree()** can be used to remove needless nodes: identity matrix pfSCS nodes, single child pfGroup nodes, and the like.
- Uses bounding spheres around nodes for fast culling—the intersection test for spheres is much faster than that for bounding boxes. If a node’s bounding sphere doesn’t intersect the viewing frustum, there’s no need to descend further into that node. There are bounding boxes around pfGeoSets; the intersection test is more expensive but provides greater accuracy at that level.
- Provides the pfPartition node type to partition geometry for fast intersection testing. Use a pfPartition node as the parent for anything that needs intersection testing.
- Provides level-of-detail (LOD) capabilities in order to draw simpler (and thus cheaper) versions of objects when they’re too far away for the user to discern small details.
- Allows intersection performance enhancement via precomputation of polygon plane equations within pfGeoSets. This pre-computation is in the form of a traversal that is nearly always appropriate—only in cases of non-intersectable or dynamically changing geometry might these cached plane equations be disadvantageous. This optimization is

performed by **pfuCollideSetup()** using the PFTRAV_IS_CACHE bit mask value.

- Sorts pfGeoSets by graphics state in the cull process, in order to minimize state changes and flatten matrix transformations, when *libpf* creates display lists to send to the draw process (as occurs in the PFMP_CULL_DL_DRAW multiprocessing mode). This procedure takes extra time in the cull stage, but can greatly improve performance when rendering a complex scene that uses many pfGeoStates. The sorting is enabled by default; it can be turned off and on by calling the function **pfChanTravMode(chan, PFTRAV_CULL, mode)** and including or excluding the PFCULL_SORT token. See "pfChannel Traversal Modes" in Chapter 6 and "Sorting the Scene" in Chapter 6 for more information on sorting.

Application Optimizations

During the application stage, IRIS Performer

- Divides the application process into two parts: the latency-critical section (which includes everything between **pfSync()** and **pfFrame()**), where last-minute latency-critical operations are performed before the cull of the current frame can start; and the noncritical portion, after **pfFrame()** and before **pfSync()**. The critical section is displayed in the channel statistics graph drawn with **pfDrawChanStats()**
- Provides an efficient mechanism to automatically propagate database changes down the process pipeline, and provides **pfPassChanData()** for passing custom application data down the process pipeline.
- Minimizes overhead copying database changes to the cull process by accumulating unique changes and updating the cull once inside **pfFrame()**. This updated period is displayed in the MP statistics of the channel statistics graph drawn with **pfDrawChanStats()**.
- Provides a mechanism for performing database intersections in a forked process: pass the PFMP_FORK_ISECT flag to **pfMultiprocess()** and declare an intersection callback with **pfIsectFunc()**.
- Provides a mechanism for performing database loading and editing operations in a forked process, such as the DBASE process: pass the PFMP_FORK_DBASE flag to **pfMultiprocess()** and declare an intersection callback with **pfDBaseFunc()**.

Specific Guidelines for Optimizing Performance

While IRIS Performer provides inherent performance optimization, there are specific techniques you can use to increase performance even more. This section contains some guidelines and advice pertaining to database organization, code structure and style, managing system resources, and rules for using IRIS Performer.

Graphics Pipeline Tuning Tips

Tuning the graphics pipeline requires identifying and removing bottlenecks in the pipeline. You can remove a bottleneck either by minimizing the amount of work being done in the stage that has the bottleneck or, in some cases, by reorganizing your rendering to more effectively distribute the workload over the pipeline stages. This section contains specific tips for finding and removing bottlenecks in each stage of the graphics pipeline. For more information on this topic, refer to the graphics library documentation (see “The IRIS GL and OpenGL Graphics Libraries” on page xxx for information on ordering these books).

Host Bottlenecks

Here are some ways to minimize the time spent in the host stage of the graphics pipeline:

- Function calls, loops, and other programming constructs require a certain amount of overhead. To make such constructs cost-effective, make sure they do as much work as possible with each invocation. For instance, drawing a pfGeoSet of triangle strips involves a nested loop, iterating on strips within the set and triangles within each strip; it therefore makes sense to have several triangles in each strip and several strips in each set. If you put only two triangles in a pfGeoSet, you’ll spend all that loop overhead on drawing those two triangles, when you could be drawing many more with little extra cost. The channel statistics can display (as part of the graphics statistics) a histogram showing the percentage of your database that is drawn in triangle strips of certain lengths.

- Only bind vertex attributes that are actually in use. For example, if you bind per-vertex colors on a set of flat-shaded quads, the software will waste work by sending those colors to the graphics pipeline, which will ignore them. Similarly, it's pointless to bind normals on an unlit pfGeoSet.
- Nonindexed drawing has less host overhead than indexed drawing because indexed drawing requires an extra memory reference to get the vertex data to the graphics pipeline. This is most significant for primitives that are easily host-limited, such as independent polygons or very short triangle strips. However, indexed drawing can be very helpful in reducing the memory requirements of a very large database.
- Enable state sorting for pfChannels (this is the default). By sorting, the CPU does not need to examine as many pfGeoStates. The graphics channel statistics can be used to report the pfGeoSet-to-pfGeoState drawing ratio.

Transform Bottlenecks

A transform bottleneck can arise from expensive vertex operations, from a scene that's typically drawn with many very tiny polygons, from a scene modeled with inefficient primitive types, or from excessive mode or transform operations. Here are some tips on reducing such bottlenecks:

- Connected primitives will have better vertex rates than independent primitives, and quadrilaterals are typically much more efficient in vertex operations than independent triangles are.
- The expensive vertex operations are generally lighting calculations. The fastest lighting configuration is always one infinite light. Multiple lights, local viewing models, and local lights have (in that order) dramatically increasing cost. Two-sided lighting also incurs some additional transform cost. On some graphics platforms, texturing and fog can add further significant cost to per-vertex transformation. The channel graphics statistics will tell you what kinds of lights and light models are being used in the scene.
- Matrix transforms are also fairly expensive, and definitely more costly than one or two explicit scale, translate, or rotate operations. When possible, flatten matrix operations into the database with **pfFlatten()**.
- The most frequent causes of mode changes are **shademodel()** (or **glShadeModel()**), textures, and object materials. The speed of these

changes depends on the graphics hardware; however, material changes do tend to be expensive. Sharing materials among different objects can be increased with the use of **pfMtlColorMode()** that is PFMTL_CMODE_AD by default. However, on some older graphics platforms (such as the **Elan, Extreme,** and **VGX**), the use of **pfMtlColorMode()** (which actually calls the IRIS GL function **lmcolor()** or the OpenGL function **glColorMaterial()**) has some associated per-vertex cost and should be used with some caution.

- If you cull stage is not a bottleneck, make sure your pfChannels sort the scene by graphics state. Even if you are running in single process mode, the extra time taken to sort the database is often more than offset by the savings in rendering time. See “Sorting the Scene” in Chapter 6 for more details on how to configure sorting.

Fill Bottlenecks

Here are some methods of dealing with fill-stage bottlenecks:

- One technique to hide the cost of expensive fill operations is to fill the pipeline from the back forward so that no part is sitting idle waiting for an upstream stage to finish. The last stage of the pipeline is the fill stage, so by drawing backgrounds or clearing the screen via **pfClearChan()** first, before **pfDraw()**, you can keep the fill stage busy. In addition, if you have a couple of large objects that reliably occlude much of the scene, drawing them very early on can both fill up the back-end stage and also reduce future fill work, because the occluded parts of the scene will fail a z-buffer test and will not have to write *z* values to the z-buffer or go on to more complex fill operations.
- Use the pfStats fill statistics (available for display through the channel statistics) to visualize your depth complexity and get a count of how many pixels are being computed each frame.
- Be aware of the cost of any fill operations you use and their relative cost on the relevant graphics hardware configuration. Quick experiments you can do to test for a fill limitation include:
  - Rendering the scene into a smaller window, assuming that doing so will not otherwise affect the scene drawn (a non-zero pfChannel LOD scale will cause a change in object LODs when you shrink the window.)
  - Using **pfOverride()** to force the disabling of an expensive fill mode

If either of these tests causes a significant reduction in the time to draw the scene, then a significant fill bottleneck probably exists.

Note: Some features may invoke expensive modes that need to be used with caution on some hardware platforms. **pfAntialias()** and **pfTransparency()** enable blending if multisampling isn't available. Globally enabling these functions on machines without multisampling can produce significant performance degradation due to the use of blending for an entire scene. Blending of even opaque objects incurs its full cost. **pfDecal()** may invoke stenciling (particularly if you have requested the decal mode PFDECAL_BASE_HIGH_QUALITY or if there is no faster alternative on the current hardware platform), which can cause performance degradations on some system. **pfFeature()** can be used to verify the availability and speed of these features on the current graphics platform.

- The cost of specific fill operations can vary greatly depending on the graphics hardware configuration. As a rule of thumb, flat shading is much faster than Gouraud shading because it reduces both fill work and the host overhead of specifying per-vertex colors. Z-buffering is typically next in cost, and then stencil and blending. On a **RealityEngine**, the use of multisampling can add to the cost of some of these operations, specifically z-buffering and stenciling. See "Multisampling" on page 476 for more information. Texturing is considered free on **RealityEngine** and **Impact** systems but is relatively expensive on a **VGX** and is actually done in the host stage on lower-end graphics platforms, such as **Extreme** and **XZ**. Some of the low-end graphics platforms also implement z-buffering on the host.
- You may not be able to achieve benchmark-level performance in all cases for all features. For instance, if you frequently change modes and you use short triangle strips, you get much less than the peak triangle mesh performance quoted for the machine. Fill rates are sensitive to both modes, mode changes, and polygon sizes. As a general rule of thumb, assume that fill rates average around 70% of peak on general scenes to account for polygon size and position as well as for pipeline efficiency.

Process Pipeline Tuning Tips

These simple tips will help you optimize your IRIS Performer process pipeline:

- Use **pfMultiprocess()** to set the appropriate process model for the current machine.
- You usually shouldn't specify more processes with **pfMultiprocess()** than there are CPUs on the system. The default multiprocess mode (PFMP_DEFAULT) attempts an optimal configuration for the number of unrestricted CPUs. However, if there are fewer processors than significant tasks (consider APP, CULL, DRAW, ISECT, DBASE) you will want experiment with the different two-process models to find the one that will yield the best overall frame rate. Use of **pfDrawChanStats()**, described in Chapter 12, "Statistics," will greatly help with this task.
- Put only latency-critical tasks between the **pfSync()** and **pfFrame()** calls. For example, put latency-critical updates, like changes to the viewpoint, after **pfSync()** but before **pfFrame()**. Put time-consuming tasks, such as intersection tests and system dynamics, after **pfFrame()**.
- You will also want to refer to the IRIX REACT™ documentation for setting up a real-time system.
- For maximum performance, use the IRIS Performer utilities in *libpfutil* for setting non-degrading priorities and isolating CPUs (**pfuPrioritizeProcs()**, **pfuLockDownProc()**, **pfuLockDownApp()**, **pfuLockDownCull()**, **pfuLockDownDraw()**). These facilities require that the application runs with root permissions. The source code for these utilities is in */usr/share/Performer/src/lib/libpfutil/lockcpu.c*. For an example of there use, see the sample source code in */usr/share/Performer/src/pguide/libpf/C/bench.c*. For more information about priority scheduling and real-time programming, see the chapter of the *IRIX System Programming Guide* entitled "Using Real-Time Programming Features." and the IRX REACT™ technical report.
- Make sure you aren't generating any floating-point exceptions. Floating-point exceptions can cause an individual computation to incur tens of times its normal cost. Furthermore, a single floating point exception can lead to many further exceptions in computations that use the exceptional result and can even propagate performance degradation down the process pipeline. IRIS Performer will detect and tell you about the existence of floating point exceptions if your

pfNotifyLevel() is set to PFNFY_INFO or PFNFY_DEBUG. You can then run your application in dbx and your application will stop when it encounters an exception, enabling you to trace the cause.

- Minimize the amount of channel data allocated by **pfAllocChanData()** and call **pfPassChanData()** only when necessary to reduce the overhead of copying the data. Copying pointers instead of data is often sufficient.

Cull Process Tips

Here are a couple of suggestions for tuning the cull process:

- The default channel culling mode enables all types of culling. If your cull process is your most expensive task, you may want to consider doing less culling operations. When doing database culling, always use view-frustum culling (PFCULL_VIEW), and usually use graphics library mode database sorting (PFCULL_SORT) and pfGeoSet culling (PFCULL_GSET) as well:

  ```
  pfChanTravMode(chan, root,
      PFCULL_VIEW | PFCULL_GSET | PFCULL_SORT);
  ```

  A cull-limited application might realize a quick gain from turning off pfGeoSet culling. If you think your database has few textures and materials, you might turn off sorting. However, if possible it would be better to try improving cull performance by improving database construction. “Efficient Intersection and Traversals” on page 460 discusses optimizing cull traversals in more detail.

- Look at the channel culling statistics for:
  - A large amount of the database being traversed by the culling process and being trivially rejected as not being in the viewing frustum. This can be improved with better spatial organization of the database.
  - A large number of database nodes being non-trivially culled. This can be improved with better spatial organization and breakup of large pfGeodes and pfGeoSets.
  - A surprising number of LODs in their fade state (the fade computations can be expensive, particularly if channel stress management has been enabled).

- Balance the database hierarchy with the scene complexity: the depth of the hierarchy, the number of pfGeoStates, and the depth of culling. See “Balancing Cull and Draw Processing with Database Hierarchy” on page 462 for details.
- pfNodes that have significant evaluation in the cull stage include pfBillboards, pfLightPoints, pfLightSources, and pfLODs.

Draw Process Tips

Here are some suggestions specific to the draw process:

- Minimize host work done in the draw process before the call to **pfDraw()**. Time spent before the call to **pfDraw()** is time that the graphics pipeline is idle. Any graphics library (or X) input processing or mode changes should be done after **pfDraw()** to take effect the following frame.
- Use only one pfPipe per hardware graphics pipeline and preferably one pfPipeWindow per pfPipe. Use multiple channels within that pfPipeWindow to manage multiple views or scenes. It’s fairly expensive to simultaneously render to multiple graphics windows on a single hardware graphics pipeline and is not advisable for a real-time application.
- Pre-define and pre-load all of the textures in the database into hardware texture memory by using a **pfApplyTex()** function on each pfTexture. You can do this texture-applying in the **pfConfigStage()** draw callback or (for multipipe applications to allow parallelism) the **pfConfigPWin()** callback. This approach avoids the huge performance impact that results when textures are encountered for the first time while drawing the database and must then be downloaded to texture memory. Utilities are provided in *libpfutil* to apply textures appropriately; see the **pfuDownloadTexList()** routine in the distributed source code file */usr/share/Performer/src/lib/libpfutil/tex.c*. The *perfly* application demonstrates this; see the perfly source file *generic.c* in either the C-language (*/usr/share/Performer/src/sample/C/common*) or C++ language (*/usr/share/Performer/src/sample/C++/common*) versions of *perfly*.
- Minimize the use of pfSCSs and pfDCSs and nodes with draw callbacks in the database since aggressive state sorting is kept local to subtrees under these nodes.

- Don't do any graphics library input handling in the draw process. Instead, use X input handling in an asynchronous process. IRIS Performer provides utilities for asynchronous input handling in *libpfutil* with source code provided in */usr/share/Performer/src/lib/libpfutil/input.c*. For a demonstration of asynchronous X input handling, see provided sample applications, such as *perfly*, and also the distributed sample programs */usr/share/Performer/src/pguide/libpf/C/motif.c* and */usr/share/Performer/src/pguide/libpfui/C/motifxformer.c*.

Efficient Intersection and Traversals

Here are some tips on optimizing intersections and traversals:

- Use pfPartition nodes on pieces of the database that will be handed to intersection traversal. These nodes impose spatial partitioning on the subgraph beneath them, which can dramatically improve the performance of intersection traversals.

  Note: Subgraphs under pfDCS, pfLOD, pfSwitch, and pfSequence nodes are not partitioned so intersection traversals of these subgraphs will not be affected.

- Use intersection caching. For static objects, enable intersection caching at initialization—first call **pfNodeTravMask()**, specifying intersection traversal (PFTRAV_ISECT), and then include PFTRAV_IS_CACHE in the mode for intersections. You can turn this mode on and off for dynamic objects as appropriate.

- Use intersection masks on nodes to eliminate large sections of the database when doing intersection tests. Note that intersections are **sproc()**-safe in the current version of IRIS Performer; you can check intersections in more than one process.

- Bundle segments for intersections with bounding cylinders. You can pass as many as 32 segments to each intersection request. If the request contains more than a few segments and if the segments are close together, the traversal will run faster if you place a bounding cylinder around the segments using **pfCylAroundSegs()** and pass that bounding cylinder to **pfNodeIsectSegs()**. The intersection traversal will use the cylinder rather than each segment when testing the segments against the bounding volumes of nodes and pfGeoSets.

Database Concerns

Optimizing your databases can provide large performance increases.

libpr Databases

The following tips will help you achieve peak performance when using *libpr*:

- Minimize the number of pfGeoStates by sharing as much as possible.
- Initialize each mode in the global state to match the majority of the database, in order to set as little local state for individual pfGeoStates as possible.
- Use triangle strips wherever possible; they produce the largest number of polygons from a given number of vertices, so they use the least memory and are drawn the fastest of the primitive types.
- Use the simplest possible attribute bindings and use flat-shaded primitives wherever possible. If you're not going to need an object's attributes, don't bind them—anything you bind will have to be sent to the pipeline with the object.
- Flat-shaded primitives and simple attribute bindings reduce the transformation and lighting requirements for the polygon. Note that the flat-shaded triangle-strip primitive renders faster than a regular triangle strip, but you have to change the index by two to get the colors right (that is, you need to ignore the first two vertices when coloring) See "Attribute Bindings" in Chapter 10 for more information.
- Use nonindexed drawing wherever possible, especially for independent polygon primitives and short triangle strips.
- When building the database, avoid fragmentation in memory of data to be rendered. Minimize the number of separate data and index arrays. Keep the data and index arrays for pfGeoSets contiguous and try to keep separate pfGeoSets contiguous to avoid small, fragmented **pfMalloc()** memory allocations.
- The ideal size of a pfGeoSet (and of each triangle strip within the pfGeoSet) depends a great deal on the specific CPU and system architecture involved; you may have to do benchmarks to find out what's best for your machine. For a general rule of thumb, use at least 4 triangles per strip on any machine, and 8 on most. Use 5 to 10 strips in each pfGeoSet, or a total of 24 to 100 triangles per pfGeoSet.

libpf Databases

When you're using *libpf*, the following tips can improve the performance of database tasks:

- Use **pfFlatten()**, especially when a pfScene contains many small instanced objects and billboards. Use **pfdCleanTree()** and (if application considerations permit) **pfdFreezeTransforms()** to minimize the cull traversal processing time and maximize state sorting scope.
- Initialize each mode in the scene pfGeoState to match the majority of the database, in order to set as little local state for individual pfGeoStates as possible. The utility function **pfdMakeSharedScene()** provides an easy to use mechanism for this task.
- Minimize the number of very small pfGeoSets (that is, those containing four or fewer total triangles). Each tiny pfGeoSet means another bounding box to test against if you're culling down to the pfGeoSet level (that is, when PFCULL_GSET is set with **pfChanTravMode()**) as well as another item to sort during culling. (If your pfGeoSets are large, on the other hand, you should definitely cull down to the pfGeoSet level.)
- Be sparing in the use of pfLayers. Layers imply that pixels are being filled with geometry that is not visible. If fill performance is a concern, this should be minimized in the modeling process by cutting layers into their bases when possible. However, this will produce more polygons which require more transform and host processing so it should only be done if it will not greatly increase database size.
- Make the hierarchy of the database spatially coherent so that culling will be more accurate and geometry that is outside the viewing frustum will not be drawn. (See Figure 6-3 for an example of a spatially organized database.)

Balancing Cull and Draw Processing with Database Hierarchy

Construct your database to minimize the draw-process time spent traversing and rendering the culled part of the database without the cull-process time becoming the limiting performance factor. This process involves making tradeoffs as a simpler cull means a less efficient draw stage. This section describes these tradeoffs and some good rules to follow to give you a good start.

If the cull and draw processes are performed in parallel, the goal is to minimize the larger of the culling and drawing times. In this case, an application can spend approximately the same amount of time on each task. However, if both culling and drawing are performed in the same process, the goal is to optimize the sum of these two times, and both processes must be streamlined to minimize the total frame time. Important parameters in this optimization include the number of pfGeoSets, the average branching factor of the database hierarchy, and the enabled channel culling traversal modes. The **pfDrawChanStats()** function (see Chapter 12, “Statistics”) can easily provide diagnostic information to aid in this tuning.

The average number of immediate children per node can directly affect the culling process. If most nodes have a high number of children, the bounding spheres are more likely to intersect the viewing frustum and all those nodes will have to be tested for visibility. At the other extreme, a very low number of children per node will mean that each bounding sphere test can only eliminate a small part of the database and so many nodes may still have to be traversed. A good place to start is with a quad-tree type organization where each node has about four children and the bounding geometry of sibling nodes are adjacent but have minimal intersection. In the ideal case, projected to a two-dimensional plane on the ground, the spatial extent of each node and its parents would form a hierarchy of boxes.

The transition from pfGeodes to pfGeoSets is an important point in the database structure. If there are many very small pfGeoSets within a single pfGeode, not culling down to pfGeoSets can actually improve overall frame time because the cost of drawing the tiny pfGeoSets may be small relative to the time spent culling them. Adding more pfGeodes to break up the pfGeoSets can help by providing a slightly more accurate cull at less cost than considering each pfGeoSet individually. In addition, pfGeodes are culled by their bounding spheres, which is faster than the culling of pfGeoSets which are culled by their bounding boxes.

The size (both spatial extend and number of triangles) can also directly impact culling and drawing performance. If pfGeoSets are relatively large, there will be fewer to cull so pfGeoSet culling can probably be afforded. pfGeoSets with more triangles will draw faster. However, pfGeoSets with larger spatial extent are more likely to have geometry that is being drawn that is outside of the viewing frustum which wastes time in the graphics stage. Breaking up some of the large pfGeoSets can improve graphics performance by allowing a more accurate cull.

With some added cost to the culling task, the use of Level-of-Detail nodes (pfLODs) can make a tremendous difference in graphics performance and image quality. LODs allow objects to be automatically drawn with a simplified version when they are in a state that yields little contribution to the scene (such as being far from the eyepoint). This allows you to have many more objects in your scene than if you always were drawing all objects at full complexity. However, you do not want the cull to be testing all LODs of an object every frame when only one will be used. Again, proper use of hierarchy can help. pfLODs (non-fading) can be inserted into the hierarchy with actual object pfLODs grouped beneath them. If the parent LOD is out of range for the current viewpoint, the child LODs will never be tested. The pfLODs of each object can be placed together under a pfGroup so that no LOD tests for the object will be done if the object is outside of the viewing frustum.

Calling **pfFlatten()**, **pfdFreezeTransforms()**, or **pfdCleanTree()** to remove extraneous nodes can often help culling performance. Use **pfFlatten()** to de-instance and apply pfSCS node transformations to leaf geometry—resulting in less work during the cull traversal. This will allow both better database sorting for the draw, and also better caching of matrix and bounding information which can speed up culling. When these scene graph modifications are not acceptable, you may reduce cull time by turning off culling of pfGeoSets but this will directly impact rendering performance by forcing the rendering of geometry that is outside the viewing frustum.

Tip: Making the scene into a graphics library object in the draw callback can show the result of the cull, which can give a visual check of what is actually being sent to the graphics subsystem. Check for objects that are far from the viewing frustum, which can indicate that the pfGeodes or pfGeoSets need to be broken up. Additionally, the rendering time of the GL object should be compared to the **pfDraw()** rendering time to see if the pfGeoSets have enough triangles in them to not incur host overhead. Alternately, the view frustum can be made larger than that used in the cull to allow simple *cull volume visualization* during real-time simulation. The IRIS Performer sample program *perfly* supports this option. Press the **z** key while in *perfly* to enable cull volume visualization and inspect the resulting images for excessive off-screen geometric content. Such content is a clear sign that the database could profitably be further subdivided into smaller components.

The code fragment in Example 13-1, taken from the sample program */usr/share/Performer/src/pguide/libpf/C/bench.c*, makes an IRIS GL object and then temporarily draws that instead of calling **pfDraw()**.

Example 13-1 Drawing an Object Without Calling **pfDraw()**

```
if (SharedFlags->glObject == MAKE_GL_OBJECT)
{
    static have_obj = 0;

    fprintf(stderr, "Making object\n");
    if (have_obj)
        delobj(1);
    makeobj(1); /* OpenGL: glNewList() */
    pfDraw();
    closeobj(); /* OpenGL: glEndList() */
    SharedFlags->glObject = DRAW_GL_OBJECT;
    have_obj = 1;
}
else if (SharedFlags->glObject == DRAW_GL_OBJECT)
    callobj(1); /* OpenGL: glCallList() */
else if (SharedFlags->glObject == PERF_DRAW)
    pfDraw();
```

Graphics and Modeling Techniques to Improve Performances

On machines with fast texture mapping, texture should be used to replace complex geometry. Complex objects, such as trees, signs, and building fronts, can be effectively and efficiently represented by textured images on single polygons in combination with applying **pfAlphaFunc()** to remove parts of the polygon that don't appear in the image. Using texture to reduce polygonal complexity can often give both an improved picture and improved performance. This is because

- The image texture provides scene complexity, and the texture hardware handles scaling of the image with MIP-map interpolation functions for minification (and, on **RealityEngine** systems, Sharpen and DetailTexture functions for magnification).
- Using just a few textured polygons rather than a complex model containing many individual polygons reduces system load.

In order to represent a tree or other 3D object as a single textured polygon, IRIS Performer can rotate polygons to always face the eyepoint. An object of

this type is known as a billboard and is represented by a pfBillboard node. As the viewer moves around the object, the textured polygon rotates so that the object appears three-dimensional. For more information on billboards, see “pfBillboard Nodes” in Chapter 5.

To determine if the current graphics platform has fast texture mapping, look for a PFQFTR_FAST return value from:

```
pfFeature(PFQFTR_TEXTURE, &ret);
```

pfAlphaFunc() with a function PFAF_GEQUAL and a reference value greater than zero can be used whenever transparency is used to remove pixels of low contribution and avoid their expensive processing phase.

Special Coding Tips

For maximum performance, routines that make extensive use of the IRIS Performer linear algebra routines should use the macros defined in *prmath.h* to allow the compiler to in-line these operations.

Use single- rather than double-precision arithmetic where possible and avoid the use of short-integer data in arithmetic expressions. Append an ‘f’ to all floating point constants used in arithmetic expressions.

BAD — In this example, values in both of the expressions involving the floating point variable *x* are promoted to double precision when evaluated:

```
float x;
if (x < 1.0)
    x = x + 2.0;
```

GOOD — In this example, both of the expressions involving the floating point variable *x* remain in single-precision mode, because there is an ‘f’ appended to the floating point constants:

```
float x;
if (x < 1.0f)
    x = x + 2.0f;
```

Performance Measurement Tools

Performance measurement tools can help you track the progress of your application by gathering statistics on certain operations. IRIS Performer provides run-time profiling of the time spent in parts of the graphics pipeline for a given frame. The **pfDrawChanStats()** function displays application, cull, and draw time in the form of a graph drawn in a channel; see Chapter 12, “Statistics,” for more information on that and related functions. There are advanced debugging and tuning tools available from Silicon Graphics that can be of great assistance. WorkShop product in the CASEVision™ tools provides a complete development environment for debugging and tuning of host tasks. The Performance Co-Pilot™ helps you to tune host code in a real-time environment. There is also the WindView™ product from WindRiver that works with IRIX REACT to do full system profiling in a real-time environment. However, progress can be made with the basic tools that are in the IRIX development environment: *prof*, *pixie*, and *glprof*. The IRIS GL debugging utility, *gldebug*, can also be used to aid in performance tuning, as can the OpenGL equivalent, *ogldebug*. This section briefly discusses getting started with these tools.

Note: See the graphics library manuals, available from Silicon Graphics, for complete instructions on using these graphics tools. See the *IRIX System Programming Guide* to learn more about *pixie* and *prof*.

Using *pixie* and *prof* to Measure Performance

You can use the IRIX performance analysis utilities *pixie* and *prof* to tune the application process. Use *pixie* for basic-block counting and use *prof* for *program counter* (PC) sampling. PC sampling gives run-time estimation of where significant amounts of time are spent, whereas basic-block counting will report the number of times a given instruction is executed.

To isolate statistics for the application process, even in single-process models, run the application through *pixie* or *prof* in APP_CULL_DRAW mode to separate out the process of interest. Both *pixie* and *prof* can generate statistics for an individual process.

When using IRIS Performer DSO libraries with *prof* you may want to provide the **-dso** option to *prof* with the full pathname of the library of interest to have IRIS Performer routines included in the analysis. When using *pixie* you

will need to have the.pixie versions of the DSO libraries in your LD_LIBRARY_PATH. Additionally, you will need a.pixie version of the loader DSO for your database in your LD_LIBRARY_PATH. You may have to pixie the loader DSO separately since pixie will not find it automatically if your executable was not linked with it. When using *prof* to do PC sampling, link with unshared libraries exclusively and use the –p option to *ld*. Then set the environment variable PROFDIR to indicate the directory in which to put profiling data files.

When profiling, run the program for a while so that the initialization will not be significant in the profiling output. When running a program for profiling, run a set number of frames and then use the automatic exit described below.

Using *gldebug* and *ogldebug* to Observe Graphics Calls

You can use the graphics utilities *gldebug* (IRIS GL) and *ogldebug* (OpenGL) to both debug and tune IRIS Performer applications. The application must be run in single-process mode in order to use *gldebug* but *ogldebug* can handling multiprocessed programs.

Use *gldebug* or *ogldebug* to:

- Show which graphics calls are being issued
- Look for frequent mode changes, or unnecessary mode settings that can be caused if your initialization of the global state doesn't match the majority of the database
- Look for unnecessary vertex bindings such as unneeded per-vertex colors, or normals for a flat-shaded object

Follow these steps to examine one frame of the application in a *gldebug* session:

1. Start up profiler of choice:

   `IRIS% gldebug -i ignore -s -F your_prog_name prog_options`

   OPEN% `ogldebug your_prog_name prog_options`

2. Turn off output and breakpoints from the control panel.
3. Set a breakpoint at **swapbuffers()** or **glXSwapBuffers()**.

4. Click the “Continue” button and go to the frame of interest.
5. Turn on breakpoints.

   Execution stops at **swapbuffers()** (or **glXSwapBuffers()**).
6. Turn on all trace output.
7. Click the “Continue” button.

   Execution stops at the next **swapbuffers()**, outputting one full scene to *GLdebug.history* (or *progname.pid.trace* for *ogldebug*).
8. Quit and examine the output.

Note: Since IRIS Performer avoids unnecessary mode settings, recording one frame shows modes that are set during that frame, but it doesn’t reflect modes that were set previously. It’s therefore best to have a program that can come up in the desired location and with the desired modes, then grab the first two frames: one for initialization, and one for continued drawing.

Using *glprof* to Find Performance Bottlenecks

You can use the IRIS GL graphics-profiling utility *glprof* to estimate where there are graphics bottlenecks so that you can tune the database. You don’t have to relink the program or run it in single-process mode to use *glprof*.

Use *glprof* to:

- See if certain views or individual objects are inherently fill- or transform-limited
  - If a scene or object is fill-limited, there can be more complex geometry in the LOD(s).
  - If a scene or object is transform-limited, you need to simplify or add LOD(s), especially for small objects.
- See significant mode changes
- Get additional scene graphics state and fill statistics such as the number of pixels fixed with polygons of different sizes and different modes

Note that since only *glprof* simulates the graphics pipeline, it may not always be entirely accurate in predicting performance.

You can use predraw callbacks on nodes to output *glprof_object* tags that will appear in a *glprof* trace. This method has the disadvantage of turning off sorting, which may increase the number of mode changes and also the balance of bottlenecks in the scene because it may change the drawing order. However, it has the advantage of giving per-object drawing statistics, and it indicates whether a specific object is fill- or transform-limited.

Example 13-2 shows sample code for a traversal that installs and removes the *glprof* object tags taken from *trav.c*. The example demonstrates general-user traversal code, then uses this traversal to install and remove *glprof* callbacks. This code is simplified from that found in */usr/src/Performer/src/lib/libpfutil/trav.c*.

Example 13-2 General Traversal

```
void
InitMyTraverser(MyTraverser *trav)
{
    trav->preFunc = NULL;
    trav->postFunc = NULL;
    trav->mstack = NULL;
    trav->data = NULL;
    trav->node = NULL;
    trav->depth = 0;
}

/* handle return value for pruning or terminating */
#define PFU_DO_RET(_ret) \
    switch (_ret) \
    { \
    case PFTRAV_PRUNE: \
        trav->node = prevNode; \
        if (needPop)           \
            pfPopMStack(trav->mstack); \
        return PFTRAV_CONT; \
    case PFTRAV_TERM: \
        trav->node = prevNode; \
        if (needPop)           \
            pfPopMStack(trav->mstack); \
        return PFTRAV_TERM; \
    }

int
MyTraverse(pfNode *node, MyTraverser *trav)
```

```
{
    int         i;
    int         numChild = 1;
    int         ret = PFTRAV_CONT, needPop = 0;
    pfNode      *prevNode = trav->node;

    if (node == NULL)
    {
        pfNotify(PFNFY_WARN, PFNFY_USAGE,
                 “MyTraverse() Null node”);
        return PFTRAV_CONT;
    }
    /*
     * for SCS and DCS push the transform on the stack
     */
    if (pfIsOfType(node, pfGetSCSClassType()) &&
        trav->mstack)
    {
        pfMatrix        mat;

        pfGetSCSMat((pfSCS *)trav->node, mat);

        pfPushMStack(trav->mstack);
        pfPreMultMStack(trav->mstack, mat);
        needPop = 1;
    }

    /* call pre-traversal callback */
    trav->node =  (pfNode *)node;
    if (trav->preFunc)
        ret = (*trav->preFunc)(trav);

    PFU_DO_RET(ret);

    /* after preFunc, in case topology changed */
    if (pfIsOfType(node, pfGetGroupClassType()))
    {
        numChild = pfGetNumChildren(node);
        if (pfIsOfType(node, pfGetGroupClassType()))
            for (i = 0 ; i < numChild ; i++)
            {
                trav->depth++;
                ret = MyTraverse((pfNode*)pfGetChild(group,
                    i), trav);
                trav->depth--;
```

```
                PFU_DO_RET(ret);
            }
        }

        PFU_DO_RET(ret);

        /* call post traversal callback */
        trav->node = node;
        if (trav->postFunc)
            ret = (*trav->postFunc)(trav);

        PFU_DO_RET(ret);

        if (needPop)
            pfPopMStack(trav->mstack);

        return PFTRAV_CONT;
    }

    /*********************************************************
     * Traversals and callbacks for installing and removing
     * pre-draw callbacks (pfNodeTravFuncs) for generating GL
     * Prof tags during drawing
     * WARNING: removes any existing pre or post-draw callbacks
     *********************************************************/

    /*
     * glprof pre-draw traversal callback
     * issues glprof_object calls
     */
    static int
    cbGLProf(pfTraverser *trav, void *data)
    {
        const pfNode        *node = pfGetTravNode(trav);
        const char          *nn;
        static char         name[80];

        if (node != NULL &&
            (nn = pfGetNodeName(node))
        {
            /* if it exists, use the node name for the tag */
            strncpy(name, nn, 79);
        }
        else
        {
```

```
        /* otherwise, use the node type string for the tag */
        strncpy(name, pfGetTypeName((pfObject *)node), 79);
    }
    glprof_object(name);
    return 0;
}

/*
 * callback for placing glprof callbacks as
 * pre-draw callbacks on nodes
 */
static int
cbPutGLProfTag(MyTraverser *trav)
{
    pfNode *node = trav->node;
    if (node != NULL &&
       ((pfIsOfType(node, pfGetGeodeClassType()))
         pfNodeTravFuncs(node, PFTRAV_DRAW, cbGLProf, NULL);
    return PFTRAV_CONT;
}

/*
 * callback to remove the pre-draw glprof tag callbacks
 */
static int
cbRmGLProfTag(MyTraverser *trav)
{
    pfNode *node = trav->node;
    if (node != NULL &&
        (pfIsOfType(node, pfGetGeodeClassType()) ||
         pfIsOfType(node, pfGetGroupClassType()))
            pfNodeTravFuncs(node, PFTRAV_DRAW, NULL, NULL);
    return PFTRAV_CONT;
}

/* glprof object tag traversal */
void
DoGLProfTraversal(pfNode *node, int mode)
{
    MyTraverser trav;
    InitMyTraverser(&trav);

    pfNotify(PFNFY_INFO, PFNFY_PRINT,
             “doing travGLProf: mode = %d”, mode);
```

```
    if (mode)   /* place glprof tag callbacks */
        trav.preFunc = cbPutGLProfTag;
    else        /* remove the callbacks */
        trav.preFunc = cbRmGLProfTag;

    MyTraverse(node, &trav);
}
```

Use this traverser in a program with a code fragment like that in Example 13-3.

Example 13-3 Using the Traverser

```
{
    /* mode = 1 - install glprof tag node callbacks
     * mode = 0 - remove glprof tag node callbacks
     */
    DoGLProfTraversal((pfNode *)scene, DBGT_GLPROF, mode);
}
```

Guidelines for Debugging

This section lists some general guidelines to keep in mind when debugging IRIS Performer applications.

Shared Memory

Because **malloc()** doesn't allocate memory until that memory is used, core dumps may occur when arenas don't find enough disk space for paging. The IRIX kernel can be configured to actually allocate space upon calling **malloc()**, but this change is pervasive and has performance ramifications for **fork()** and **exec()**. Reconfiguring the kernel is not recommended, so be aware that unexplained core dumps can result from inadequate disk space.

Be sure to initialize pointers to shared memory and all other nonshared global values before IRIS Performer creates the additional processes in the call to **pfConfig()**. Values put in global variables initialized after **pfConfig()** will only be visible to the process that set them.

For detailed information about other aspects of shared memory, see "Memory Allocation" in Chapter 10.

Use the Simplest Process Model

When debugging an application that uses a multiprocess model, first use a single-process model to verify the basic paths of execution in the program. You don't have to restructure your code; simply select single-process operation by calling **pfMultiprocess(PFMP_APPCULLDRAW)** to force all tasks to initiate execution sequentially on a frame-by-frame basis.

If an application fails to run in multiprocess mode but runs smoothly in single-process mode, you may have a problem with the initialization and use of data that's shared among processes.

If you need to debug one of multiple processes, use

```
IRIS% dbx -p progname
```

while the process is running. This will show the related processes and allow you to choose a process to trace. The application process will always be the process with the lowest process id. In order after that will be the (default) clock process, then the cull process, and then the draw.

Once the program works, experiment with the different multiprocess models to achieve the best overall frame rate for a given machine. Don't specify more processes than CPUs. Use **pfDrawChanStats()** to compare the frame timings of the different stages and frame times for the different process models.

Avoid Floating-Point Exceptions

Arrange error handling for floating-point operations. To see floating-point errors, turn debug messages on and enable floating-point traps. Set **pfNotifyLevel(PFNFY_DEBUG)**.

The goal is to have no NaN (Not a Number), INF (infinite value), or floating-point exceptions resulting from numerical computations.

Notes on Tuning for RealityEngine Graphics

This section contains some specific notes on performance tuning with RealityEngine graphics.

Multisampling

Multisampling provides full-scene antialiasing with performance sufficient for a real-time visual simulation application. However, it isn't free and it adds to the cost of some other fill operations. With RealityEngine graphics, most other modes are free until you add multisampling— multisampling requires some fill operations to be performed on every subpixel. This is most noticeable with z-buffering and stenciling operations, but also applies to **blendfunction()** and **glBlendFunc()**. Texturing is an example of a fill operation that can be free on a RealityEngine and isn't affected by the use of multisampling.

The multisampling hardware reduces the cost of subpixel operations by optimizing for pixels that are fully opaque. Pixels that have several primitives contributing to their result are thus more expensive to evaluate and are called *complex pixels*. Scenes usually end up having a very low ratio of complex pixels.

Multisampling offers an additional performance optimization that helps balance its cost: a virtually free screen clear. Technically, it doesn't really clear the screen but rather allow you to set the *z* values in the framebuffer to be undefined. Therefore, use of this clear requires that every pixel on the screen be rendered every frame. This clear is invoked with a pfEarthSky using the **PFES_TAG** option to **pfESkyMode()**. Refer to the pfEarthSky(3pf) reference page for more detailed information.

Transparency

There are two ways of achieving transparency on a RealityEngine: blending, and screen-door transparency with multisampling.

Blended transparency, using the IRIS GL **blendfunction()** routine (or the OpenGL **glBlendFunc()** equivalent), can be used with or without

multisampling. Blending doesn't increase the number of complex pixels, but is expensive for complex pixels.

To reduce the number of pixels with very low alpha, one can use a **pfAlphaFunc()** that ignores pixels of low alpha, such as alpha less than 3 or 4. This will slightly improve fill performance and probably not have a noticeable effect on scene quality. Many scenes can use values as high as 60 or 70 without suffering degradation in image quality. In fact, for a scene with very little actual transparency, this can reduce the fuzzy edges on textures that simulate geometry (such as trees and fences) that arise from MIP-mapping.

Screen-door transparency gives order-independent transparent effects and is used for achieving the fade-LOD effect. It's a common misperception that screen-door transparency on RealityEngine gives you *n* levels of transparency for *n* multisamples. In fact, *n* samples gives you 4*n* levels of transparency, because RealityEngine uses 2-pixel by 2-pixel dithering. However, screen-door transparency causes a dramatic increase in the number of complex pixels in a scene, which can affect fill performance.

Texturing

Texturing is free on a RealityEngine if you use a 16-bit texel internal texture format. There are 16-bit texel formats for each number of components. These formats are used by default by IRIS Performer but can be set on a pfTexture with **pfTexFormat()**. Using a 32-bit texel format will yield half the fill rate of the 16-bit texel formats.

Do not use huge ranges of texture coordinates on individual triangles. This can incur both an image quality degradation and a severe performance degradation. Keep the maximum texture coordinate range for a given component on a single triangle under $(1 << (13-\log_2(\mathit{TexCSize}))$ where *TexCSize* is the size in the dimension of that component.

The use of Detail Texture and Sharpen can greatly improve image quality. Minimize the number of different detail and sharpen splines (or just use the internal default splines). Applying the same detail texture to many base textures can incur a noticeable cost when base textures are changed. Detail textures are intended to be derived from a high-resolution image that corresponds to that of the base texture.

Other Tips

Two final notes on RealityEngine performance:

- Changing the width of antialiased lines and points is expensive.
- **pfMtlColorMode()** (which calls the IRIS GL function **lmcolor()** or the OpenGL function **glColorMaterial()**) has a huge performance benefit.

Chapter 14

“Programming with C++”

This chapter discusses the differences in programming using the C and C++ programming interfaces.

Chapter 14

Programming with C++

This chapter provides an overview of some of the differences between programming IRIS Performer using the C++ Application Programming Interface (C++ API) rather than the C-language Application Programming Interface (C API) which is described in the earlier chapters of this guide.

Overview

Although this guide uses the C API throughout, the C++ API is in every way equal and in some cases superior in functionality and performance to the C API.

Every function available in the C API is available in the C++ API. All of the C API routines tightly associated with a class have a corresponding member function in the C++ API, e.g. **pfGetDCSMat()** becomes **pfDCS::getMat()**. Routines not closely associated with a class are the same in both APIs. Examples include high-level global functions such as **pfMultiprocess()** and **pfFrame()** and low-level graphics functions such as **pfAntialias()**.

Most of the routines associated with a class can be divided into three categories: setting an attribute, getting attribute and acting on the object. In the C API, sets were usually expressed as pf<Class><Attribute>, gets as pfGet<Class><Attribute> and simple actions as pf<Action><Class>, where <Class> is the abbreviation for the full name of the class. In some cases where there was no room for confusion or this usage was awkward, the routine names were shortened, e.g. **pfAddChild()**.

The principal difference in the naming of member functions in the C++ API and the corresponding routine name in t he C-language API is in the naming of member functions where the “pf” prefix and the <Class> identifier are dropped. In addition, the word “set” or “get” is prefixed when attribute values are being set or retrieved. Hence, value setting functions are usually

have names of the form pfClass::set<Attribute>, value getting functions are named pfClass::get<Attribute>, and actions appear as pfClass::<Action>.

Table 14-1 Corresponding routines in the C and C++ API

C Routine	C++ Member Function	Description
pfMtlColor()	pfMaterial::setColor()	Set material color
pfGetMtlColor()	pfMaterial::getColor()	Get material color
pfApplyMtl()	pfMaterial::apply()	Apply the material

Note: Member function whose names begin with “pf_”, “pr_” or “nb_” are internal functions and should not be used by applications.

Class Taxonomy

There are three main types of C++ classes in IRIS Performer. The following description is based on this categorization of the main types: *public structs*, *libpr classes*, and *libpf classes*. A fourth distinct class is pfType, the class used to represent the type of *libpr* and *libpf classes*.

Public Structs

These classes are public structs with exposed data members. They include pfVec2, pfVec3, pfVec4, pfMatrix, pfQuat, pfSeg, pfSphere, pfBox, pfCylinder and pfSegSet.

libpr Classes

These classes derive from pfMemory. When multiprocessing, all processes share the same copy of the object’s data members.

libpf Classes

These classes derive from pfUpdatable and when multiprocessing, each APP, CULL and ISECT process has a unique copy of the object’s. data members.

pfType Class

As with the C API, information about the class hierarchy is maintained with pfType objects.

Programming Basics

Header Files

The C++ include files for *libpf* and *libpr* are in */usr/include/Performer/pf* and */usr/include/Performer/pr*, respectively. An application using a class should include the corresponding header file.

Table 14-2 Header Files for *libpf* Scene Graph Node Classes

***libpf* Class**	**Include File**
pfBillboard	<Performer/pf/pfBillboard.h>
pfDCS	<Performer/pf/pfDCS.h>
pfGeode	<Performer/pf/pfGeode.h>
pfGroup	<Performer/pf/pfGroup.h>
pfLOD	<Performer/pf/pfLOD.h>
pfLayer	<Performer/pf/pfLayer.h>
pfLightPoint	<Performer/pf/pfLightPoint.h>
pfLightSource	<Performer/pf/pfLightSource.h>
pfMorph	<Performer/pf/pfMorph.h>
pfNode	<Performer/pf/pfNode.h>
pfPartition	<Performer/pf/pfPartition.h>
pfSCS	<Performer/pf/pfSCS.h>
pfScene	<Performer/pf/pfScene.h>
pfSequence	<Performer/pf/pfSequence.h>

Table 14-2 (continued) Header Files for *libpf* Scene Graph Node Classes

***libpf* Class**	**Include File**
pfSwitch	<Performer/pf/pfSwitch.h>
pfText	<Performer/pf/pfText.h>

Table 14-3 Header Files for Other *libpf* Classes

***libpf* Class**	**Include File**
pfBuffer	<Performer/pf/pfBuffer.h>
pfChannel	<Performer/pf/pfChannel.h>
pfEarthSky	<Performer/pf/pfEarthSky.h>
pfLODState	<Performer/pf/pfLODState.h>
pfPipe	<Performer/pf/pfPipe.h>
pfPipeWindow	<Performer/pf/pfPipeWindow.h>
pfTraverser pfPath	<Performer/pf/pfTraverser.h>

Table 14-4 Header Files for *libpr* Graphics Classes

***libpr* Class**	**Include File**
pfColortable	<Performer/pr/pfColortable.h>
pfDispList	<Performer/pr/pfDispList.h>
pfFog	<Performer/pr/pfFog.h>
pfFont	<Performer/pr/pfFont.h>
pfGeoSet pfHit	<Performer/pr/pfGeoSet.h>
pfGeoState	<Performer/pr/pfGeoState.h>
pfHighlight	<Performer/pr/pfHighlight.h>
pfLPointState	<Performer/pr/pfLPointState.h>

Table 14-4 (continued) Header Files for *libpr* Graphics Classes

***libpr* Class**	**Include File**
pfLight pfLightModel	<Performer/pr/pfLight.h>
pfMaterial	<Performer/pr/pfMaterial.h>
pfSprite	<Performer/pr/pfSprite.h>
pfState	<Performer/pr/pfState.h>
pfString	<Performer/pr/pfString.h>
pfTexture pfTexGen pfTexEnv	<Performer/pr/pfTexture.h>

Table 14-5 Header Files for Other *libpr* Classes

***libpr* Class**	**Include File**
pfCycleBuffer pfCycleMemory	<Performer/pr/pfCycleBuffer.h>
pfDataPool	<Performer/pr/pfDataPool.h>
pfFile	<Performer/pr/pfFile.h>
pfSphere pfBox pfCylinder pfPolytope pfFrustum pfSeg pfSegSet	<Performer/pr/pfGeoMath.h>
pfVec2 pfVec3 pfVec4 pfMatrix pfQuat pfMatStack	<Performer/pr/pfLinMath.h>
pfList	<Performer/pr/pfList.h>

Table 14-5 (continued) Header Files for Other *libpr* Classes

***libpr* Class**	**Include File**
pfMemory	<Performer/pr/pfMemory.h>
pfObject	<Performer/pr/pfObject.h>
pfStats	<Performer/pr/pfStats.h
pfType	<Performer/pr/pfType.h
pfWindow	<Performer/pr/pfWindow.h

Creating and Deleting IRIS Performer Objects

The IRIS Performer base classes all provide **operator new** and **operator delete.** All *libpr* and *libpf* objects must be explicitly created with **operator new** and deleted with **operator delete**. Objects of these classes cannot be created statically, on the stack or in arrays.

The default new operator creates objects in the current shared memory arena, if one exists. *libpr* objects and public structs have an additional new operator that takes an arena argument. This new operator allows allocation from the heap (indicated by an arena of NULL) or from a shared memory arena created by the application with IRIX **acreate()**.

Example 14-1 Legal Creation of Objects in C++

```
// legal creation of libpf objects
pfDCS    *dcs = new pfDCS;         // only way

// legal creation of libpr objects
pfGeoSet *gs = new pfGeoSet;       // from default arena
pfGeoSet *gs = new(NULL) pfGeoSet; // from heap

// legal creation of public structs
pfVec3 *vert = new pfVec3;           // from default arena
pfVec3 *verts = new pfVec3[10];      // array from default
static pfVec3 vert(0.0f, 0.0f, 0.0f); // static
```

Example 14-2 Illegal Creation of Objects in C++

```
// illegal creation of libpf objects
pfDCS    *dcs = new(NULL) pfDCS;    // not in shared mem
pfDCS    *dcs = new pfDCS[10];      // array

// illegal creation of libpr objects
pfGeoSet *gs = new pfGeoSet[10];    // array

// illegal creation of public structs
pfVec3 *vert = new(NULL) pfVec3[10];// array, non-default new
```

Caution: This last item in Example 14-2 is illegal because C++ does not provide a mechanism to delete arrays of objects allocated with a new operator defined to take additional arguments, e.g. **operator new(size_t s, void *arena)**. Attempting to delete an array of objects allocated in this manner can cause unpredictable and fatal results such as the invocation of the destructor a large number of times on pointers inside and outside of the original allocation.

All objects of classes derived from pfObject or pfMemory are reference counted and must be deleted using **pfDelete()**, rather than the delete operator. **pfDelete()** checks the reference count of the object and when multiprocessing, delays the actual deletion until other processes are done with the object. To decrement the reference count and delete with a single call use **pfUnrefDelete()**.

Public structs such as pfVec3, pfSphere, etc. may be deleted either with **pfDelete()** or the delete operator.

Invoking Methods on IRIS Performer Objects

Since *libpr* and *libpf* objects are allocated, they can only be maintained by reference.

Passing Vectors and Matrices to Other Libraries

Passing arrays of floats is very common in graphics programming. Calls to IRIS GL or OpenGL often require an array of floats or a matrix. In the C API,

the data types such as pfMatrix are arrays and so can be passed straight through to OpenGL routines, e.g.

```
pfMatrix ident;
pfMakeIdentMat(ident);
glLoadMatrix(ident);
```

In the C++ API, the data field of the pfMatrix must be passed instead, e.g.

```
pfMatrix ident;
ident.makeIdent();
glLoadMatrix(ident.mat);
```

Porting from C API to C++ API

When compiled with C++, IRIS Performer supports three usages of the API:

1. Pure C++ API. This is the default style of usage.
2. Pure C API. This can be achieved by defining the token PF_CPLUSPLUS_API to be 0, e.g. by adding the line:

   ```
   #define PF_CPLUSPLUS_API 0
   ```

   in source files before they include any IRIS Performer header files. In this mode all data types are the same as when compiling with C.
3. C++ API and C API. This mode can be enabled by defining the token PF_C_API to be 1, e.g. by adding the line

   ```
   #define PF_C_API 1
   ```

   in source files before they include any IRIS Performer header files. In this mode, both C++ and C functions are available and data types are C++. See the section below concerning passing certain data types.

Typedefed Arrays vs. Structs

In the C API, the pfVec2, pfVec3, pfVec4, pfMatrix and pfQuat data types are all typedefed arrays. In the C++ API, they are all structs. When converting C code to use the C++ API or when compiling CAPI code with both APIs enabled, be sure to change routines in your code that pass objects of these types. In the C++ API, you almost always want to pass arguments of these types by reference rather than by value.

For example, the C API routine

```
void MyVectorAdd(pfVec2 dst, pfVec2 v1, pfVec2 v2)
{
    dst[0] = v1[0] + v2[0];
    dst[1] = v1[1] + v2[1];
}
```

should be rewritten for the C++ API to pass by reference

```
void MyVectorAdd(pfVec2& dst, pfVec2& v1, pfVec2& v2)
{
    dst[0] = v1[0] + v2[0];
    dst[1] = v1[1] + v2[1];
}
```

or

```
void MyVectorAdd(pfVec2* dst, pfVec2* v1, pfVec2* v2)
{
    dst->vec[0] = v1->vec[0] + v2->vec[0];
    dst->vec[1] = v1->vec[1] + v2->vec[1];
}
```

Without this change, time will be wasted copying v1 and v2 by value and the result will not be returned to the routine calling **MyVectorAdd().**

Interface Between C and C++ API Code

The same difference in passing conventions applies if you are calling a C function from code that uses the C++ API. Functions passing typedefed arrays with the C API must have a different prototype for use with the C++ API. Macros for use in C prototypes bilingual can be found in */usr/include/Performer/prmath.h*.

```
#if PF_CPLUSPLUS_API
#define PFVEC2 pfVec2&
#define PFVEC3 pfVec3&
#define PFVEC4 pfVec4&
#define PFQUAT pfQuat&
#define PFMATRIX pfMatrix&
#else
#define PFVEC2 pfVec2
#define PFVEC3 pfVec3
```

```
#define PFVEC4 pfVec4
#define PFQUAT pfQuat
#define PFMATRIX pfMatrix
#endif /* PF_CPLUSPLUS_API */
```

These macros are used in the C API prototypes for IRIS Performer that pass typedefed arrays, e.g.

```
extern float pfDotVec2(const PFVEC2 v1, const PFVEC2 v2);
```

But they are not necessary or appropriate for when passing pointers to typedefed arrays in C, e.g.

```
extern void pfFontCharSpacing(pfFont *font, int ascii,
    pfVec3 *spacing);
```

because a pointer to a struct is passed in the same manner as a pointer to an array.

Subclassing pfObjects

With the C API, the main mechanism for extending the functionality of the classes provided in IRIS Performer is the specification of the user data pointer on pfObjects with **pfUserData()** and the specification of callbacks on pfNodes with **pfNodeTravFuncs()** and **pfNodeTravData()**. The C++ API also supports these mechanisms, but also provides the additional capacity to subclass new data types from the classes defined in IRIS Performer. Subclassing allows additional member data fields and functions to be added to IRIS Performer classes. At it's simplest, subclassing merely provides a way of adding additional data fields that is more elegant than hanging new data structures off of a pfObject's user data pointer. But in some uses, subclassing also allows significantly more control over the functional behavior of the new object because virtual functions can be overloaded to bypass, replace or augment the processing handled by the parent class from IRIS Performer.

Initialization and Type Definition

The new object should provide two static functions, a constructor that initializes the instances pfType* and a static data member for the type system as shown in the following table;

Table 14-6 Data and Functions Provided by User Subclasses

Class Data or Function	Function
static void init()	Initialize the new class
static pfType* getClassType()	Returns the pfType* of the new class
static pfType* classType	Stores the pfType* of the new class
constructor	Sets the pfType* for each instance

The **init()** member function should initialize any data structures that are related to the class as a whole, as opposed to any particular instance. The most important of these is the entry of the class into the type system. For example, the Rotor class defined in the Open Inventor loader (see *Rotor.h* and *Rotor.C* in */usr/share/Performer/src/lib/libpfdb/libpfiv*) is a subclass of pfDCS. It's initialization function merely enters the class into the type system.

Example 14-3 Class Definition for a Subclass of pfDCS

```
public Rotor : public pfDCS
{
    static void init();
    static pfType* getClassType(){ return classType; };
    static pfType* classType;
}

pfType *Rotor::classType = NULL;

Rotor::Rotor()
{
    setType(classType);  // set the type of this instance
    ...
}

void
Rotor::init()
```

```
{
    if (classType == NULL)
    {
        pfDCS::init();
        classType =
            new pfType(pfDCS::getClassType(), “Rotor”);
    }
}
```

As described in the section below, the initialization function, Rotor::init() should be called before **pfConfig()**.

Defining Virtual Functions

Below is the example of the Rotor class which specifies the traversal function for the *libpf* application traversal. When overloading a traversal function, it is usually desirable to invoke the parent class function, in this case, **pfDCS::app()**. It is not currently possible to overload *libpf*'s intersection or culling traversals. See “libpf Objects and Multiprocessing” on page 495.

Example 14-4 Overloading the *libpf* Application Traversal

```
int
Rotor::app(pfTraverser *trav)
{
    if (enable)
    {
    pfMatrix mat;
        double now = pfGetFrameTimeStamp();

        // use delta and renorm for large times
        prevAngle += (now - prevTime)*360.0f*frequency;
        if (prevAngle > 360.0f)
            prevAngle -= 360.0f;
        mat.makeRot(prevAngle, axis[0], axis[1], axis[2]);
        setMat(mat);
        prevTime = now;
    }

    return pfDCS::app(trav);
}

int
```

```
Rotor::needsApp(void)
{
    return TRUE;
}
```

The same behavior could also be implemented in either the C or C++ IRIS Performer API using a callback function specified with **pfNodeTravFuncs()**.

Note: Classes of pfNodes that need to be visited during the application traversal even in the absence of any application callbacks should define the virtual function **needsApp()** to return TRUE.

Accessing Parent Class Data Members

Accesses to parent class data is made through the functions on the parent class. Data members on built-in classes should never be accessed directly.

Multiprocessing and Shared Memory

Initializing Shared Memory

In general to assure safe multiprocess operation with any DSOs providing C++ virtual functions or defining new pfTypes. Initialization should be carried out in the following sequence:

1. Call **pfInit()**. This initializes the type system and for *libpf* applications sets up shared memory.
2. Initialize any application-supplied classes:

   a) Load any application-specific C++ DSOs

   b) Call **pfdInitConverter()** to initialize and load any converter DSOs.

   c) Enter any user-supplied pfTypes into the type system, e.g. call

   ```
   Rotor::init()
   ```
3. Call **pfConfig()**. This forks off other processes as specified by **pfMultiprocess()**
4. Create *libpf* and *libpr* objects.

Note: Pure *libpr* applications that do their own multiprocessing outside of IRIS Performer with **fork()** should explicitly create shared memory with **pfInitArenas()** before calling **pfInit()**. Otherwise, the type system will not be visible in the address space of other processes.

More on Shared Memory and the Type System

Advanced

IRIS Performer objects or other objects that use pfTypes can only be shared between *related processes*. Related processes are those created with **fork()** or **sproc()** from the main process after **pfInit()** in a *libpf* application, e.g. processes created by **pfConfig()**.

New pfTypes should be added before **pfConfig()** forks off other processes so that the static data member containing the class type is visible in all processes, otherwise pf<Class>::getClassType() will return NULL in other processes. This effectively precludes the creation of subclasses of IRIS Performer objects after **pfConfig()**.

Advanced

Virtual Address Spaces and Virtual Functions

When using virtual functions, it's very important that the object code reside at the same address in all processes. Normally, this is not an issue since the object code for all IRIS Performer classes is loaded (whether statically linked or loaded as dynamic shared objects, DSOs) before **pfConfig()** is called to fork off processes. For user-defined C++ classes with virtual functions, it's important that the object code reside at the same virtual address space in all processes that access them. For this reason, the DSOs for any user-defined classes should be loaded before **pfConfig()** regardless of whether they use the pfType system or not.

Data Members and Shared Memory

Non-static Member Data

The default operator new for objects derived from pfObject causes all instances to be created in shared memory, so that objects will be visible to other related processes that need to see them.

Static Member Data

Classes having static data members that may change value and need to be visible from all processes, should allocate shared memory for the data (e.g. pfMalloc or new pfMemory) and set the static data member to point to this memory before **pfConfig()** as shown in the following example.

Example 14-5 Changeable Static Data Member

```
class Rotor : public pfDCS
{
    static int* instanceCount;
}
Rotor::instanceCount = NULL;

void Rotor::init()
{
...
    instanceCount = new(sizeof(int)) pfMemory;
    *instanceCount = 0;
}
Rotor::Rotor()
{
...
    (*instanceCount)++; // increment the creation counter
}
```

A static data member whose value is set before **pfConfig()** and never changes thereafter does not need to be allocated from shared memory. The **classType** member of Rotor is an example of this since the class should be initialized, i.e. **Rotor::init()** called, before **pfConfig()**.

libpf Objects and Multiprocessing

Advanced

The multiprocessing behavior of *libpf* objects (i.e. those deriving from pfNode or pfUpdatable) differs from that of *libpr* objects. Both are typically created in shared memory, but with a *libpr* object, all processes share the same data members, while *libpf* objects have a built in multiprocessing data mechanism that provides different copies in the APP, CULL and ISECT stages of the IRIS Performer pipeline. The term *multibuffering* refers to the maintenance and frame-accurate updating of these data.

Advanced

With a user-defined subclass of a *libpf* class, the original data elements of the *libpf* parent class are still multibuffered. However, the parallel multibuffer copies maintained in the other processes are instances of the parent class rather than the subclass. This is not normally visible to the application, since even for callbacks in the CULL and ISECT processes, the application always works from the pointer to the copy used in the APP process, in part so that objects can be identified by comparison of pointers. However, this difference would be visible if the virtual traversal functions for culling or intersection were overloaded. These virtual functions should not be overloaded by the subclass since they will not have any effect when the CULL or ISECT stages are in separate processes. Node callbacks specified with **pfNodeTravFuncs()** should be used instead.

Subclassing will be vastly simplified and more flexible in a future release.

Performance Hints

Constructor Overhead

It's quite natural to frequently construct and destroy arrays of public structs such as pfVec3 on the stack. Beware, even though the constructors for these classes are empty, it still requires a function call for each element of the array. The same applies to classes such which contain arrays of structs, e.g. pfSegSet contains an array of pfSegs.

Math Operators

Assignment operators, e.g. "+=", are significantly faster than their corresponding binary operators, e.g. "+" because the latter involves constructing a temporary object for the return value.

Appendix A

Image Gallery

This appendix contains images created using IRIS Performer and various scene models.

Appendix A

Image Gallery

This appendix contains views of some of the models that come with IRIS Performer. The images in this chapter were created using the Lightscape Visualization System, available from Lightscape Technologies, Inc. in San Jose, California. For information on Lightscape software, call 408-246-1155.

Figure A-1 Simulated view of an atrium

The image in Figure A-1 was created by A.J. Diamond, Donald Schmitt and Company, Toronto. For information, call 416-862-8800. The database that the image illustrates is part of the IRIS Performer software distribution.

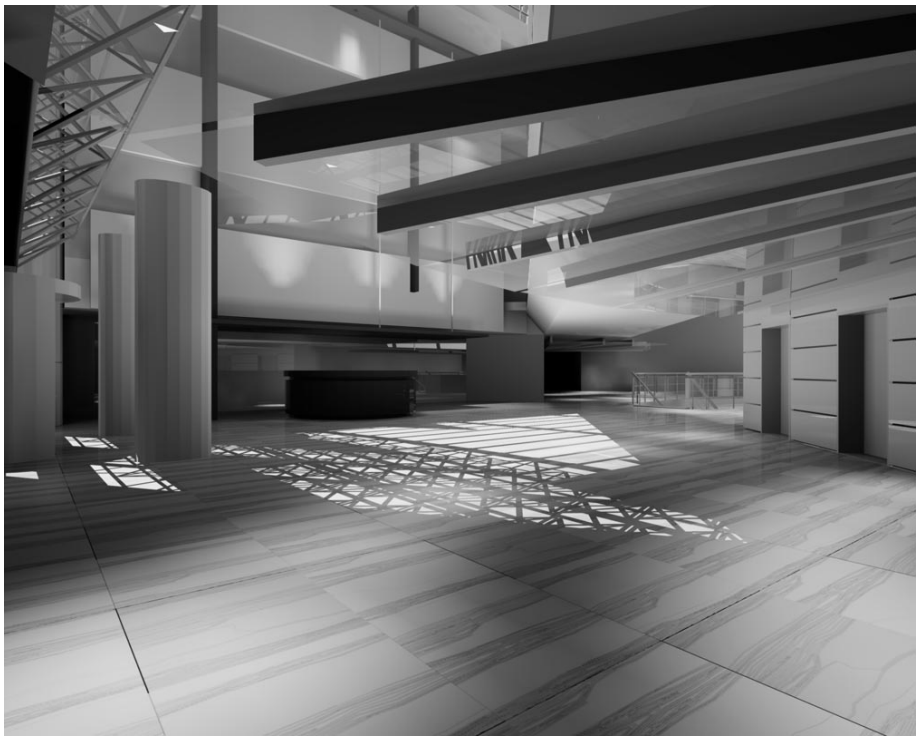


Figure A-2 Another simulated view of the atrium

The image in Figure A-2 was also created by A.J. Diamond, Donald Schmitt and Company, from the same database.

Figure A-3 Simulated view of a castle

The image in Figure A-3 was created by Advanced Graphics Applications, Toronto. For more information, call 905-279-3838. The database that the image illustrates is part of the Friends of Performer software distribution.

Figure A-4 Simulated hallway view

The image in Figure A-4 was created by A.J. Diamond, Donald Schmitt and Company.

Figure A-5 Simulated hotel lobby

The image in Figure A-5 was created by Design Vision Inc., Toronto. For more information, call 416-585-2020.

Figure A-6 Simulated waiting room

The image in Figure A-6 was created by Digital Architecture, Isao Nagaoka and Joe Henke, New York. For information, call 212-587-4148.

Figure A-7 Simulated conference room

The image in Figure A-7 was created by Advanced Graphics Applications.

Figure A-8 Parliament stairway

The image in Figure A-8 was created by A.J. Diamond, Donald Schmitt and Company.

Figure A-9 Unity Temple interior

The image in Figure A-9 was created by Lightscape Technologies, Inc. The database that the image illustrates is a model of the Unity Church and community house project designed by Frank Lloyd Wright in 1906. This database is part of the IRIS Performer software distribution.

Glossary

alias

An alternate extension for a file type as processed by the **pfdLoadFile()** utility. For example, VRML "*.wrl*" files are in a sense an alias for Open Inventor "*.iv*" files since the Open Inventor loader can read VRML files as well. Once the alias is established, files with alternate extensions will be loaded by the designated loader.

application buffer

The main (and usually only) buffer of *libpf* data structures such as the nodes in the scene graph. Alternate buffers may be created and data can be constructed in these new buffers from parallel processes to support high-performance asynchronous database paging during real-time simulation.

arena

An area (allocation area) from which shared memory is allocated. Usually the arena is the default one created by **pfInit()** or **pfInitArenas()**, but some objects (e.g. those in libpr) may be created in any arena returned by **acreate().** IRIS Performer calls that accept an arena pointer as an argument can also accept the NULL pointer, indicating that the memory should be allocated from the heap. See also *heap*.

asynchronous database paging

An advanced method of scene-graph creation, asynchronous database paging allows desired data to be read from a disk or network connection and IRIS Performer internal data structures to be built for this data using one or more processes running on separate CPUs rather than performing these tasks in the application process. Once the data structures are created in these database processes, they must be explicitly merged into the application buffer.

attribute binding

The binding of an attribute specifies how often an attribute is specified and the scope of each specification. For example given a collection of triangles for rendering, a color can be specified with each vertex of each triangle, with each triangle or once for the entire collection of triangles.

base geometry

The object with the lowest visual priority in a pfLayer node's list of children. This would be the runway, for example, in a runway and stripes airport database example. See also *layer geometry*.

bezel

The beveled border region surrounding any item, but most notably, around the edge of a CRT monitor.

billboard

Geometry that rotates to follow the eyepoint. This is often simply a single texture mapped quadrilateral used to represent an object that has roughly cylindrical or spherical symmetry, such as a tree or a puff of smoke, respectively. IRIS Performer supports billboards that can rotate about an axis for cylindrical objects or a point for spherical objects.

binning

The action of the sort phase of libpf's cull traversal that segregates drawable geometry into major sections (such as opaque and transparent) before the per-bin sorting based in the contents of associated pfGeoStates, so that state changes can be reduced by drawing groups of similar geometry sequentially while still drawing semitransparent objects in the desired order within the frame.

bins

The unique collections into which the cull traversal segregates drawable geometry. The number of bins is defined by calls to **pfChanBinSort()** and **pfChanBinOrder()**. Typical bins are those for opaque and transparent geometry, where opaque objects are rendered first for superior image quality when using blended transparency.

bounding volume

A convex region that encompasses a geometric object or a collection of such objects. IRIS Performer pfGeoSets have axis-aligned bounding boxes which are rectangular boxes whose faces are along the X, Y, or Z axes. Each IRIS Performer pfGeode has a bounding sphere that contains the bounding box of each pfGeoSet in the pfGeode. Performer group nodes have hierarchical bounding spheres that contain (bound) the geometry in their descendent nodes. The purpose of bounding volumes is to allow a quick test of a region for being off-screen or out of range of intersection search vectors.

buffer scope

All IRIS Performer nodes are created in a pfBuffer, either the primary application buffer or in an alternate returned by **pfNewBuffer()**. Only one pfBuffer is considered "current," and this buffer can be selected using **pfSelectBuffer()**. All new nodes are created in the current pfBuffer and will be visible *only* in the current buffer until that pfBuffer is merged into the application buffer using **pfMergeBuffer()**. A process cannot access nodes that do not have scope in its current buffer except through the special "buffer" commands: **pfBufferAddChild()**, **pfBufferRemoveChild()**, and **pfBufferClone()**. Thus they are said to have buffer scope.

channel

A visual channel specifies how a geometric scene should be rendered to the display device. This includes the viewport area on the screen as well as the location, orientation and field of view associated with the viewer or camera.

channel group

A set of channels that share attributes such as the eyepoint or callbacks. When a shared attribute is set on any member of the group, all members get the new value. Channel groups are most commonly used for adjacent displays making up a panorama.

channel share mask

A bit mask indicating the attributes that are shared by all channels in a *channel group*. Typical shared attributes are field of view, view specification, near and far clipping distances, the scene to be drawn, stress parameters, level of detail parameters, the earth/sky model, and swapbuffer timing.

children

IRIS Performer's hierarchical scene graph of pfNodes has internal nodes derived from the pfGroup class, and each node attached below a pfGroup type node is known as a child of that node. The complete list of child nodes are collectively termed the children of that node.

class hierarchy

The provenance through which IRIS Performer classes are defined. This class hierarchy defines the data elements and member functions of these data types through the notion of class inheritance as described below.

class inheritance

Class inheritance describes the process of defining one object as a special version of another. For example, a pfSwitch node is a special version of a pfGroup node in that it has control information about which children are active for drawing or intersection. In all other respects, a pfSwitch has the same capabilities as a pfGroup, and the Performer API supports this notion directly in both the C and C++ API by allowing a pfSwitch node to be used wherever a pfGroup is called for in a function argument. This same flexibility is supported for all derived types.

clipped

Geometry is said to be clipped when some or all of its geometric extent crosses one or more clipping planes and the portion of the geometry beyond the clipping plane is mathematically trimmed and discarded.

clipping planes

The normal clipping planes are those that define the viewing frustum. These are the left, right, top, bottom, near, and far clipping planes. All rendered geometry is clipped to the intersection of the half-spaces defined by these planes and only the portion inside all six is displayed by the graphics hardware.

cloned instancing

The style of instancing that creates a (possibly partial) copy of a node hierarchy rather than simply making a reference to the parent node. This allows pfDCS nodes and other internal nodes to be changed in the copy without changing those in the original. Also see *shared instancing*.

cloning

Making a copy of a data structure recursively copying down to some specified level. In IRIS Performer **pfCopy()** creates a shallow copy. **pfClone()** creates a deeper copy that creates new copies of internal nodes, but not of leaf nodes. This means that the pfDCS, pfSwitch, and other internal nodes in the cloned hierarchy are separate from those in the original.

compiled mode

IRIS Performer pfGeoSets are designed for rapid immediate-mode rendering and in most situations outperform IRIS GL and OpenGL display list usage. In those cases where GL display lists are desired, pfGeoSets may be placed in compiled mode, whereby a GL display list will be created the first time the pfGeoSet is rendered and this display list will be used for subsequent renderings until the pfGeoSet compiled mode flag is explicitly reset. Once a pfGeoSet is compiled, any changes to its data arrays by the pfMorph node or other means will not be effective until the compiled-mode flag is cleared.

complex pixels

Pixels for which several different geometric primitives contribute to the pixel's assigned color value. Such pixels are rare in typical scenes, and only exist at edges of polygons unless multisample blending is in use. When this blending mode is used, then all pixels rendered as neither fully opaque nor fully transparent are complex pixels.

critically damped

A closed-loop control system notion where the feedback transfer function is just right: not so slow that the system goes out of range before correction is applied and not so fast that overcorrection causes rapid swings or variation. This should be the goal of any user specified stress management function.

cull

See *culling*.

cull volume visualization

The visual display of the culling volume, usually the same as the viewing frustum, to which the scene is culled before rendering. Normally the projected culling volume fills the display area. By rendering with a larger field of view or from a eye point that differs from the origin of the frustum,

the tightness of culling can be determined for database tuning. The culling volume itself is often drawn in wireframe.

culling

Discarding database objects that are not visible. Usually this refers to discarding objects located outside the current *viewing frustum*. This is done by comparing the bounding volume of these database objects with the six planes that bound the frustum. Objects completely outside may be safely discarded. See also *occlusion culling*.

data fusion

IRIS Performer's ability to read data in a variety of different database formats and convert it into the internal IRIS Performer scene database format. Further, the ability of these different formats to provide special run-time behavior via callback functions or node subclassing and to have these different data formats all active in their native modes simultaneously

database paging

Loading databases from disk or network into memory for traversal during real-time simulation. Database paging is implicit in large area simulations due to the huge database sizes inherent in any high-resolution earth database. A frequent component of database paging is texture paging, in which new textures are downloaded to the graphics system at the same time new geometry is loaded from disk.

debug libraries

IRIS Performer libraries compiled with debugging symbols left in are known as debug libraries. These libraries provide greater and more accurate stack trace information when examining core dumps, such as during application development.

decal geometry

Objects that appear "above" other objects in pfLayer geometry. In a runways and stripes example, the stripes would be the decal geometry. There can be multiple layers of decals with successively higher visual priorities. See *layer geometry*.

depth complexity

The “pixel rendering load” of a frame which is defined as the total number of pixels written divided by the number of pixels in the image. For example, an image of two full-screen polygons would have a depth complexity of 2. It is often observed that different types of simulation images have predictable depth complexities, with values ranging from a low of 2.5 for high altitude flight simulation to 4 or more for ground-based simulations. These figures can serve as a guide when configuring hardware and estimating frame rates for visual simulation systems. IRIS Performer fill statistics provide detailed accounting and real-time visualization of depth complexity, as seen in *perfly*.

displace decaling

An implementation method for decal geometry that uses a Z-displacement to render coplanar geometry. The actual displacement used is a combination of a fixed offset and a range-based scaled offset which are combined to produce the effective offset.

display List

A list into which graphics commands are placed for efficient traversal. Both IRIS Performer and the underlying graphics libraries (OpenGL or IRIS GL) have their own display list structures.

draw mask

A bit-mask specified for both pfNodes and pfChannels which together selects a subset of the scene graph for rendering. The node and channel draw masks are logically AND-ed together during the CULL traversal which prunes the node if the result is zero. Draw masks may be used to “categorize” the scene graph where each bit represents a particular characteristic. Each node contains these masks, binary values whose bits serve as flags to indicate if the node and its children are considered drawable, intersectable, pickable, and so on. Most of these bits are available for application use.

drop

Refers to frame processing. When in locked or fixed phase and a processing stage takes too long, the frame is dropped and not rendered. Dropped frames are a sure sign of system overload.

DSO

See *dynamic shared object*.

dynamic

Something that is updated automatically when one of its attributes or children in a scene graph changes. Often refers to the update of hierarchical bounding volumes in the scene graph.

dynamic shared object

A Library which is not copied into the final application executable file but is instead loaded dynamically (that is, when the application is launched). Since DSOs are shared, only one copy of a given DSO is loaded into memory at a time, no matter how many applications are using it. DSOs also provide the dynamic binding mechanism used by the IRIS Performer database loaders.

Euler angles

A set of three angles used to represent a rotation.

See *heading*, *pitch* and *roll*.

fixed frame rate

Rendering images at a consistent chosen frame rate. Fixed frame rates are a central theme of visual simulation and are supported in IRIS Performer via the PFPHASE_LOCK and PFPHASE_FLOAT modes. Maintaining a fixed frame rate in databases of varying complexity is difficult and is the task of IRIS Performer stress processing, which changes LOD scales based in measured system load.

flatten

Flattening consists of taking multiple instances of a single object and converting them into separate objects (deinstancing) and then applying any static transformations defined by pfSCS nodes to the copied geometry; this action improves performance at a cost in memory space.

flimmering

The visual artifact associated with improperly drawing coplanar Z-buffered geometry. The term is derived from the German verb *flimmern*, which is has synonyms flitter, flicker, sparkle, twinkle, and vibrate (as in, *die Augen flimmern mir*, my eyes are swimming.) One way to understand flimmering is to consider the screen space interpolation of Z-depth values, wherein a

discrete difference of depth must be interpolated across a discrete number of pixels (or sub-pixels). When two polygons that would be coplanar in an infinite precision real-number context are considered in this discrete interpolation space, it is clear that the interpolated depth values will differ when the delta-Z to delta-pixels ratios are relatively prime. The image that results is essentially a Moirè pattern showing the modular relationship of the differences in the least significant bits of interpolated depth between the polygons. The *libpr* **pfDecal()** function and *libpf* **pfLayer()** node exist to handle the drawing of coplanar geometry without flimmering.

floating phase

The style of frame overload management where the next frame after an overloaded frame is allowed to start at any vertical retrace boundary rather than being forced to wait for a specific boundary as in the LOCKED phase.

frame

The term frame is used to mean “image” in most IRIS Performer contexts. The image being rendered by the hardware is drawn into a “frame buffer” which is simply an image memory. This image, when delivered via video signals to a monitor or projector, exists as one or two video fields. In the one-field case, also known as non-interlaced, each row of the image is read from the frame buffer and generated as video in sequential order. In the interlaced method, the first field of display comprises alternate lines, one field for the odd lines and one field for the even lines. In this mode a frame consists of two fields, as the norm for NTSC broadcast video. Also, the frame is the unit of work in Performer; the main loop in any Performer application consists of calls to **pfFrame()**.

frame accurate

In a pipelined multiprocessing model, at any particular time the different stages of the pipeline is working on different frames. Data in the pipeline is called frame accurate when a change made to the data in a particular frame is not visible in downstream stages of the pipeline until those stages begin processing that frame. Processing of libpf objects are frame accurate because multiple copies of data are retained for the different pipeline stages.

free-running

The unconstrained phase relationship of image generation where frame rendering is initiated as soon as the previous frame is complete without consideration of a minimum or maximum frame rate.

frustum

A truncated pyramid—two parallel rectangular faces, one smaller than the other, and four trapezoidal faces that connect the edges of one rectangular face with the corresponding edges of the other rectangular face. Note that it is pronounced as it is spelled and contains only one “r” despite common misuse. Also, the plural of frustum is frusta, which does not contain an “s”.

gaze vector

The +Y axis from the eyepoint—informally, the direction the eye is facing.

graph

A network of nodes connected by arcs. An IRIS Performer scene graph is so termed due to its having this form. In particular, a Performer scene graph must be an acyclic graph. See also *scene graph*.

graphics context

The set of modes and other attributes maintained by IRIS GL or OpenGL in both system software and the hardware graphics pipeline that defines how subsequent geometry is to be rendered. It is this information which must be saved and restored when drawing occurs in multiple windows on a single graphics pipeline.

graphics state elements

Individual libpr state components, such as material color, line stipple pattern, point size, current texture definition, and the other elements that comprise the graphics context.

heading

In the context of X-axis to the right, Y-axis forward, and Z-axis up, then heading is rotation about the Z-axis. This is the disturbing rotation that pivots your car clockwise or counterclockwise during a skid. Heading is also known as *yaw*, but IRIS Performer uses the term heading to keep the H, P, and R abbreviations distinct from X, Y, and Z. Also see *Euler angles*.

heap

The process heap is the normal area from which memory is allocated by **malloc()** when more memory is required, **sbrk()** is automatically called to increase the process virtual memory. Also see *arena*.

identity matrix

A square matrix with ones down the main diagonal and zeroes everywhere else. This matrix is the multiplicative identity in matrix multiplication.

immediate mode rendering

Immediate mode rendering operations are those which immediately issue rendering commands and transfer data directly to the graphics hardware rather than compiling commands and data into data structures such as display lists. See *compiled mode*.

instancing

An object in the scene is called instanced if there is more than one path through the scene graph that reaches it. Instancing is most commonly used to place the same model in more than one location by instancing it under more than one pfDCS transformation node.

intersection pipeline

Like the rendering pipeline, IRIS Performer supports a two-stage multiprocessing pipeline between the APP and ISECT processes. See also *rendering pipeline*.

latency

The amount of time between an input and the response to that input. For example rendering latency is usually defined as the time from which the eyepoint is set until the display devices scans out the last pixel of the first field corresponding to that eye point.

latency-critical

Operations which must be performed during the current frame and which will reliably finish quickly. An example of this would be reading the current position of a head-tracking device from shared memory.

layer geometry

Objects that appear “above” other objects in pfLayer geometry. In a runways and stripes example, the stripes would be the decal geometry. There can be multiple layers of decals with successively higher visual priorities. See also *base geometry*.

level of detail

The idea of representing a single object, such as a house, with several different geometric models (a cube, a simple house, and a detailed house, for example) that are designed for display at different distances. The models and ranges are designed such that the viewer is unaware of the substitutions being made. This is possible because distant objects appear smaller and thus can be rendered with less detail. The IRIS Performer pfLOD node and the associated pfLODState implement this scheme.

libpf

One of IRIS Performer's two core libraries. *libpf* manages multiprocessing and scene graph traversals. Built on top of *libpr*. Multiple copies of *libpf* objects are automatically maintained so that the APP, CULL and ISECT stages of the processing pipeline do not collide

libpfdu

IRIS Performer's database utilities library. Layered on top of *libpf* and *libpr*. Includes functions for building and optimizing geometry before putting it into a scene graph.

libpfutil

IRIS Performer's general utility library which is distributed in source form for both usage and information.

libpr

One of IRIS Performer's two core libraries. *libpr* manages graphics state and rendering, while also providing a number of math and shared memory utility functions. Provides the foundation for *libpf*. All processes share the same copy of *libpr* objects.

libpr classes

The low-level structured data types of libpr. These objects—with the exception of pfCycleBuffers—lack the special multibuffered multiprocess data exclusion support that libpf objects provide.

light point

A point of light such as a star or a runway light. Accurate display of light points requires that they attenuate and fog differently than other geometry (see *punch through*). In flight simulation, light points often have additional parameters concerning angular distributions of illumination.

load

The processing burden of rendering a frame. This includes both processing performed on the host CPU and in the graphics subsystem. It is the maximum of these times (sum in single process mode) that is used to compute the system stress level for adjusting pfLODState values.

locked phase

A style of frame overload processing where drawing may only begin on specific vertical retraces, namely those that are an integer multiple of the basic frame rate.

morph attribute

One of the collections of arrays of floating point data used in the pfMorph node's linear combination processing. This process multiplies each element of each source array by a changeable weight value for that source array and sums the result of these products to produce the destination array.

morphing

The mathematical manipulation of pfGeoSet data (positions, normals, colors, texture coordinates) to cause a shape-shifting behavior. This is very useful for animated characters, continuous terrain level of detail, for smooth object level of detail, and for a number of advanced applications. In IRIS Performer, morphing is provided by the pfMorph node.

multiple inheritance

Deriving a class from more than one other class. This is in contrast to single inheritance in which a type hierarchy is a tree. IRIS Performer does not use multiple inheritance.

multithreaded

In the context of IRIS Performer culling, multithreading is an option for increased parallelism when multiple pfChannels exist in a single IRIS Performer rendering pipeline. In this case, multiple cull processes are created to work on culling the channels of a pfPipe in parallel. For example, a single IRIS Performer pipeline stage (such as the CULL) is multithreaded when configured as multiple, concurrent processes. These "threads" are not arranged in pipeline fashion but work in parallel on the same frame.

mutual exclusion

Controlling access to a data structure so that two or more threads in a multiprocessing application cannot simultaneously access a data structure. Mutual exclusion i s often required to prevent a partially updated data structure from being accessed while it is in an invalid state.

node

An IRIS Performer libpf data object used to represent the structure of a visual scene. Nodes are either leaf nodes that contain geometry via libpr, or are internal nodes derived from pfGroup that control and define part of the scene hierarchy.

nonblocking file access

A method of obtaining data from a file without having to wait for any other processes to finish using the file. Such accesses involve a two-step transaction in which the application first indicates the task to be performed and is given a handle. This handle can later be used to inquire about the status of the file action: is it in progress, has it completed, or has there been an error.

non-degrading priorities

Process priorities are used by the operating system to decide when and for how long processes should run. A non-degrading priority specifies that the process scheduling should not take into account how long the process has been running when deciding whether to let another process run. The use of non-degrading priorities is important for real-time performance.

occlusion culling

The discarding of objects which are not visible because they are occluded by other closer objects in the scene, e.g. a city behind a mountain. See also *culling*.

opera lighting

The generic term for a powerful carbon-arc lamp producing an intense light such as that invented by John H. Kliegl and Anton T. Kliegl for use in public staged events and cinematographic undertakings that is often mounted within a dual-gimballed exoskeletal framework to afford the lamp sufficient freedom of orientation that the projected beam can be made to track and highlight performers as they move across a stage. The temperature of the

thermal plasma that develops between the carbon electrodes of such arc lamps can be determined by spectroscopic investigation of its dissociated condition, and has been found to be between 20,000°C and 50,000°C. The term can also refer to a stage-lighting technique that projects an image of a background scene onto the stage or screen. Accurate visual simulation of both of these light types (as well as common vehicle headlights, airplane landing lights, and searchlights) is provided by the projected texture capability of the pfLightSource node.

overload

A condition where the time taken to process a frame is longer than the desired frame rate allows. This causes the goal of a fixed-frame rate to be unattainable, and thus is an undesired situation.

overrun

A synonym for overload in the context of fixed frame rate rendering.

pair-wise morphing

The geometric blending of two topologically equivalent objects. Usually this is done by specifying weights for each object, e.g. 90% of object A plus 10% of object B. Each vertex in the resulting object is a linear interpolation between the vertices in the original object. See *morphing*.

parent

The IRIS Performer node directly above a given node is known as the parent node.

passthrough data

Data which is passed down the steps in the rendering pipeline until it reaches a callback. Such data provides the mechanism whereby an application can communicate information between the app, cull, and draw stages in a pipelined manner without code changes in single-CPU and multiprocessing applications.

path

A series of nodes from a scene graph's root down to a specific node defines a path to that node. When there are multiple paths to a node (and thus the scene graph is really a graph rather than a tree) this path can be important when interpreting an intersection or picking request. For example, if a car

model uses instancing for the tires, just knowing that a tire is picked is not sufficient for further processing.

perfly

The application distributed with IRIS Performer that serves as a demonstration program installed in */usr/sbin* as well as a programming example found in */usr/share/Performer/src/sample/apps/C* and */usr/share/Performer/src/sample/apps/C++* for the C and C++ versions, respectively.

phase

An application's synchronization mode—defining how the system behaves if the processing and drawing time for a given frame extends past the time allotted for a frame. See also *locked phase* and *floating phase*.

pipe

Used to refer to both an IRIS Performer software rendering pipeline and to a graphics hardware rendering pipeline, such as a RealityEngine. See *rendering pipeline*.

pitch

In the context of X-axis to the right, Y-axis forward, and Z-axis up, then pitch is rotation about the X-axis. This is the rotation that would raise or lower the nose of an aircraft. Also see *Euler angles*.

popping

The term for the highly noticeable instantaneous switch from one level of detail to the next when morph or blend transitions are not used. This problem is distracting and should be eliminated in high-quality simulation applications.

process callbacks

The mechanism through which a developer takes control of processing activities in the various IRIS Performer traversals and major processing stages: the application traversal, the cull traversal, the draw traversal, and the intersection traversal all provide a mechanism for registered process callbacks. These are user functions that are invoked at the beginning of the indicated processing stage, and in the process handling the traversal.

projective texturing

A texture technique that allows texture images to be projected onto polygons in the same manner as a slide or movie projector would exhibit keystone distortion when images are cast non-obliquely onto a wall or screen. This effect is perfect for projected headlights and similar lighting effects.

prune

To eliminate a node from further consideration during *culling*.

punch through

Decreasing the rate at which intensely luminous objects such as light points are attenuated as a function of distance. Normal fogging is inappropriate for such objects because up close they are actually much brighter than can be rendered given the dynamic range of the frame buffer and raster display devices.

reference counting

The counter within each pfObject and pfMemory object that keeps track of how many other data structures are referencing the particular instance. The primary purpose is to indicate when an object may be safely deleted because it is no longer referenced.

rendering pipeline

An IRIS Performer rendering pipeline, represented in an application by a pfPipe. Typically a rendering pipeline has three stages APP, CULL and DRA.W. These stages may be handled in separate processes or combined into one or two processes.

right-hand rule

Derived from a simple visual example for the direction of positive rotation about an axis, the right-hand rule states that the curled fingers of the right hand indicate the direction of positive rotation when the right hand is placed about the desired axis with the thumb pointing in the positive direction. The positive angle is the one that rotates the primary axes toward each other. For example, a positive rotation (counter clockwise) about the X-axis takes the positive Y-axis into the position previously occupied by the positive Z-axis.

roll

In the context of X-axis to the right, Y-axis forward, and Z-axis up, then roll is rotation about the Y-axis. This is the rotation that would raise and lower the wings of an aircraft, leading to a turn. Also see *Euler angles.*

scene

A collection of geometry to be rendered into a pfChannel.

scene complexity

The complexity of the scene for rendering purposes, in particular the amount of geometry, transformations, and graphics state changes in the scene.

scene graph

A hierarchical assembly of IRIS Performer nodes linked by explicit attachment arcs that constitutes a virtual world definition for traversal and subsequent display.

search path

A list of directory names given to IRIS Performer to specify where to look for data files which aren't specified as full path names.

sense

An indication of whether a positive angle is interpreted as representing a clockwise or counterclockwise rotation with respect to an axis. All CCW rotations in IRIS Performer are specified by positive (+) angles and negative angles represent CW rotations.

shadow map

A special texture map created by rendering a scene from the view of a light source and then recording the depth at each pixel. This Z-map is then used with projective texturing in a second pass to implement cast shadows. The entire process is automated by the IRIS Performer pfLightSource node.

shared instancing

The simplest form of instancing whereby two or more parent nodes share the same node as a child. In this situation, any change made to the child will be seen in each instance of that node. Also see *cloned instancing.*

shininess

The coefficient of specular reflectivity assigned to a pfMaterial that governs the appearance of highlights on geometry to which it is bound.

siblings

The name given to nodes that have the same parent in a scene graph.

skip

Refers to frame processing. See *drop*.

sorting

The grouping together of geometry with similar graphics state for more efficient rendering with fewer graphics state changes. IRIS Performer sorts during scene graph traversal.

spacing

The relative motion required to move the starting point for subsequent pfFont rendering after drawing a particular character pfGeoSet in a pfFont. This motion is a pfVec3 to allow arbitrary escapement for character sets that use vertical rather than horizontal text layouts. Note that for vertically oriented fonts, the origin should be such that motion by the spacing value crosses the character: in other words, the origin should be on the left for left-to-right fonts and at the top for top-to-bottom fonts, on the bottom for bottom-to-top fonts, and on the right for right-to-left fonts.

spatial organization

The grouping together of geometric objects that are spatially close to each other in the scene graph. For optimal culling performance, the scene should be organized spatially.

sprite

A transformation that rotates a piece of geometry, usually textured, so that it always faces the eye point.

stage

This is a section of the IRIS Performer software rendering pipeline and is one of application, culling, or drawing. Sometimes used to refer to either of the two non-pipeline tasks of intersection and asynchronous database processing.

state

State refers to attributes used to render an object that are managed during traversal. State commonly falls into two areas, traversal state that affects which portions of the scene graph are traversed and graphics state that affects how something is rendered. Graphics state includes the current transformation, the graphics modes managed by pfGeoStates and other state such as stenciling.

stencil decaling

An implementation method for pfLayer nodes that uses an extra bit per pixel in the frame buffer to record the Z-buffer pass or fail status of the base geometry. This bit is then used as a visibility determination (rather than the Z-buffer test) for each of the layers, which are rendered in bottom (lowest visual priority) to top (highest visual priority) order. Z-buffer updating is disabled during the stencil rendering operation, and is restored when the pfLayer node has been completely rendered. Stencil-bit processing is the highest quality mode of pfLayer operation.

stress

IRIS Performer stress processing is the closed-loop feedback mechanism that monitors cull and draw times to determine how pfLODState range scale factors should be adjusted to compensate for system load in order to maintain a chosen frame rate.

subgraph

A connected subset of a scene graph; usually, the set consisting of all descendents of a particular node.

texel

Short for “texture element”—a pixel of a texture.

texture mapping

Displaying a texture as though it were the surface of a given polygon.

tile

A section of a spatially subdivided database or a rectangular subregion of a larger texture image.

transformation

Homogeneous 4x4 matrices that define 3D transformations—some combination of scaling, rotation, and translation.

transition distance

The distance at which one level-of-detail model is switched for another. When fading or morphing between levels-of-detail, the distance at which 50% of each model is rendered. See *level of detail*.

traversals

One of IRIS Performer's pre-order visitations of a hierarchical scene graph. Traversals for application, culling, and intersection processing are internal to libpf and user-written traversals are supported by the pfuTraverser tools.

traversing

See *traversals*.

trigger routine

A routine that initiates a traversal or the invocation of a callback in another process. **pfCull()** triggers the cull traversal. **pfFrame()** triggers processing for the current frame.

up vector

The +Z axis of the eyepoint, defining the display's "up" direction. Must be perpendicular to the *gaze vector*.

view volume visualization

The display of the viewing frustum for a particular channel; usually done by rendering a wireframe version of the frustum with a different eye point or field-of-view. See *cull volume visualization*.

viewing frustum

The *frustum* containing the portion of the scene database visible from the current eyepoint.

viewpoint

The location of the camera or eye used to render the scene.

viewport

The portion of the framebuffer used for rendering. Each pfChannel has a viewport in the framebuffer of its corresponding pfPipeWindow.

visual

An construct that the X Window System uses to identify framebuffer configurations.

widget

A manipulable or decorative element of a graphical user interface. Much of the programming for GUI elements is associated with defining the reaction of widgets to user mouse and keyboard events.

window manager

A special X window system client which handles icons, window placement, and window borders and titles.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

W

X

Y

Z

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-1680-030.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389