OpenGL Performer™ Getting Started Guide

Version 3.1

007-3560-004

CONTRIBUTORS

Written by George Eckel, Ken Jones, and Tammy Domeier

Illustrated by Dany Galgani, Chrystie Danzer, and Chris Wengelski

Production by Karen Jacobson

Engineering contributions by the Performer Team, including Sharon Clay, Tom McReynolds, Don Hatch, Jenny Zhao, Remi Arnaud, Yair Kurzion, Rob Mace, Marcin Romaszewicz, Allan Schaffer, Tom Flynn, Radomir Mech, Angus Dorbie, Paolo Farinelli, and Alexandre Naaman.

New Features in This Guide

This revision of the guide documents OpenGL Performer 3.1, which has the following features:

- GPU programming on systems like Silicon Graphics Onyx4 UltimateVision visualization systems
- Curves and surfaces (higher-order primitives)
  - Parametric surfaces
  - Subdivision surfaces
- pfGeoArrays for efficient use of vertex arrays on systems like Silicon Graphics Onyx4 UltimateVision visualization systems
- New data sources
  - Maya exporter to OpenGL Performer
  - CATIA import to OpenGL Performer
- Miscellaneous
  - Small object culling
  - Linux: RedHat 8 support
  - Linux: RedHat 9 support
  - IRIX: Remove O32 support
  - Linux: Remove `gcc2` support

Record of Revision

Version	Description
001	1997 Original publication.
002	November 2000 Updated for the 2.4 version of OpenGL Performer.
003	December 2002 Updated for the 3.0 version of OpenGL Performer.
004	December 2003 Updated for the 3.1 version of OpenGL Performer.

Contents

Figures

Tables

About This Guide

Welcome to the OpenGL Performer application development environment. OpenGL Performer provides a programming interface (with ANSI C and C++ bindings) for creating real-time graphics applications and offers high-performance, multiprocessed rendering in an easy-to-use 3D graphics toolkit. OpenGL Performer interfaces with the OpenGL graphics library; this library combined with the IRIX, Linux, or MicroSoft Windows (Windows 2000, Windows NT, or Windows XP) operating system forms the foundation of a powerful suite of tools and features for creating real-time 3D graphics applications.

This guide introduces the most important concepts and classes in the Performer libraries. A full explanation of all OpenGL Performer classes can be found in the *OpenGL Performer Programmer's Guide*. Use this guide to quick-start your programming using the OpenGL Performer application programming interface (API.)

What Is OpenGL Performer?

OpenGL 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. OpenGL Performer provides a flexible, intuitive, toolkit-based solution for developers who want to optimize application performance.

Why Use OpenGL Performer?

Use OpenGL Performer to:

- Build visual simulation applications and virtual reality environments
- Render on-air broadcast and virtual set applications quickly
- View large simulation-based design tasks
- Maximize the graphics performance of any application

Applications that require real-time visuals, free-running or fixed-frame-rate display, or high-performance rendering can benefit from using OpenGL Performer.

OpenGL Performer drastically reduces the work required to tune your application's performance. General optimizations include:

- Use of highly tuned routines for all performance-critical operations
- Reorganization of graphics data and operations for faster rendering

OpenGL Performer also handles SGI architecture-specific tuning issues for you by selecting the best rendering and multiprocessing modes at run time, based on the system configuration.

OpenGL Performer is an integral part of SGI visual simulation systems. It provides the interface to advanced features available exclusively with the SGI product line, such as the Silicon Graphics Onyx4 UlitimateVision, InfiniteReality, Silicon Graphics Octane, and Silicon Graphics O2, VPro, and Impact graphics subsystems. OpenGL Performer provides the features to develop a sophisticated image generation system in a powerful, flexible, and extensible software environment. OpenGL Performer is also tuned to operate efficiently on a variety of graphics platforms; you do not need the hardware sophistication of InfiniteReality graphics to benefit from OpenGL Performer.

What You Should Know Before Reading This Guide

To use OpenGL 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" are not explained in this guide. It helps if you are familiar with the OpenGL graphics libraries.

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

What This Guide Contains

This guide is divided into the following parts, chapters, and appendices: Part One is an overview of OpenGL Performer features; Part Two is a programming overview. For more detailed programming instructions, see the *OpenGL Programmer's Guide.* If your interest is in programming only, skip to Part Two.

Part One: Overview of OpenGL Performer

- Chapter 1, "Getting Acquainted with OpenGL Performer," provides a hands-on example of an OpenGL Performer application, Perfly, to introduce you to the features of OpenGL Performer.
- Chapter 2, "OpenGL Performer Basics," provides an introduction to OpenGL Performer, including a survey of visual simulation techniques, descriptions of features and libraries, and a discussion of some of the specific details of OpenGL Performer structure and use.

Part Two: Programming with OpenGL Performer

- Chapter 3, "OpenGL Performer Programming Interface," describes the fundamental ideas behind the OpenGL Performer programming interface.
- Chapter 4, "Introduction to OpenGL Performer Concepts," describes the basic classes that implement the database-to-display pipeline.
- Chapter 5, "Creating a Display with pfChannel," discusses many of the important classes that constitute the process of taking data from a scene graph database and rendering it on a display system.
- Chapter 6, "Creating Scene Graphs," describes how to create, change, load, and save scene graphs.
- Chapter 7, "Creating Geometry with pfGeoSets," describes how to create surfaces and geometric objects.
- Chapter 8, "Specifying the Appearance of Geometry with pfState and pfGeoState," describes how to define the appearance of geometry.
- Chapter 9, "Placing Geometry in a Scene," describes how to reorient and scale geometry.
- Chapter 10, "Controlling Frame Rate," describes how to control the frame rate.

- Chapter 11, “Multiprocessing,” describes how to use multiprocessing.
- Chapter 12, “Database Paging,” describes how to page the database efficiently.
- Chapter 13, “Intersection Testing,” describes how to check for intersections.
- Chapter 14, “Creating a User Interface,” describes how to create a user interface.
- Chapter 15, “Optimizing Performance,” describes how to optimize an application.
- Appendix A, “Building a Visual Simulation Application Using libpf” follows the development of a skeleton application program that introduces you to the basic concepts involved in creating a visual simulation application with `libpf`.
- Appendix B, “Building a Visual Simulation Application Using libpfv” describes how to use the library `libpfv` to build an application using a graphical viewer.
- Appendix C "Image Gallery," contains some sample images created by using OpenGL Performer to display various scene databases.

These chapters are followed by a glossary and an index.

Conventions

This guide uses the following typographical conventions:

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

Note that in some cases it is convenient to refer to a group of similarly named OpenGL 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.

All code examples for IRIX and Linux are available in both C and C++ forms in the source directory `/usr/share/Performer/src/pguide`; on Windows the examples can be found in `%PFROOT%\Src\pguide`.

Internet and Hardcopy Reading for the OpenGL Performer Series

The OpenGL Performer series include the following in printed and online versions:

- *OpenGL Performer Programmer's Guide*
- IRIX, Linux, or Windows*OpenGL Performer Getting Started Guide* (this book)

To read these online books, point your browser at the following:

- http://docs.sgi.com

For general information about OpenGL Performer, use the following URL:

- http://www.sgi.com/software/performer

The `info-performer` mailing list provides a forum for discussion of OpenGL Performer including technical and nontechnical issues. Subscription requests should be sent to `info-performer-request@sgi.com`. Much like the `comp.sys.sgi.*` newsgroups on the Internet, it is not an official support channel but is monitored by several interested SGI employees familiar with the toolkit. The OpenGL Performer mailing list archives are located at the following URL:

- http://oss.sgi.com/projects/performer/mail/info-performer/

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  `http://docs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043-1351

We value your comments and will respond to them promptly.

PART ONE

Overview of OpenGL Performer I

Chapter 1, “Getting Acquainted with OpenGL Performer.”

Chapter 2, “OpenGL Performer Basics.”

Getting Acquainted with OpenGL Performer

This chapter provides a hands-on example of an OpenGL Performer application, Perfly, to introduce you to the features of OpenGL Performer. If you are already familiar with OpenGL Performer or visual simulation in general, you might like to skip to the overview in Chapter 2, "OpenGL Performer Basics."

Installing the Software

Follow the instructions in the OpenGL Performer release notes to install the software. This process places the appropriate libraries, header files, sample databases, man pages, online books, and demonstration programs on your system.

Note: For the IRIX operating system, use `grelnotes` to read the release notes. For the Linux operating system, the release notes are located in `/usr/doc/performer-3.0`. The release notes are located in `%PFROOT%\Doc\performer-3.0` for Windows systems.

Exploring the OpenGL Performer Sample Scenes with Perfly

This section introduces you to basic OpenGL Performer functionality through the Perfly demo application. Perfly is a basic visual simulation application that can load, store, and display scene databases in many common formats. Using the following subsections, this section describes how to use Perfly to look at several sample databases provided with OpenGL Performer:

Locations of Perfly and Example Source Code

The Perfly application is the sample OpenGL Performer application included with the installation. For the IRIX operating system, it is installed in `/usr/sbin`. For the Linux operating system, it is installed in `/usr/X11R6/bin`. On Windows systems, you can find Perfly in the directory `%PFROOT%\Bin`.

For IRIX and Linux systems, source code for Perfly is provided in `/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 in the `/usr/share/Performer/src/sample/C++` directory. On Windows systems, these locations are `%PFROOT%\Src\sample\C` and `%PFROOT%\Src\sample\C++`, respectively.

The Perfly demo is a good demonstration of OpenGL Performer in action because it is a complete application. It is, however, a large and complex piece of code. A better place to start exploring programming with OpenGL Performer is the sample code provided in `/usr/share/Performer/src/sample/pguide`. On Windows systems, this is located in `%PFROOT%\Src\pguide`. Under this directory, you can find examples for programming many of the features available in each of the libraries that make up OpenGL Performer, using either C or C++. Not all example programs appear in both directories, so you will want to look at both the C and C++ directories.

Starting and Quitting Perfly

To launch Perfly, enter the following:

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

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. A command-line entry of `perfly -h` displays a list of the command-line options.

When you want to quit Perfly, either press the `Esc` key or click the **Quit** button on the Perfly graphical user interface (GUI).

Basic Perfly Controls

The Perfly demo provides a GUI 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 do not need to learn about them before using Perfly.

You can operate the control panel using the mouse buttons and the keyboard. Many other keys on the keyboard are active and can be used to control Perfly even when the control panel is not displayed. The `perfly` man page contains a list of these keys sequences and their effects as well as details on motion models.

Looking 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 or right to turn in place; you will continue to pivot until you place the cursor back in the center of the screen.

Do not worry 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.

Approaching the Building

To approach the City Hall model, turn until you are facing it (if you are not 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 will continue gliding forward at constant speed and can hold down the middle mouse button to steer. The Perfly application shows you how the basic visual simulation tools work. This example uses a section of the New Jerusalem City Hall (see Figure 1-1).

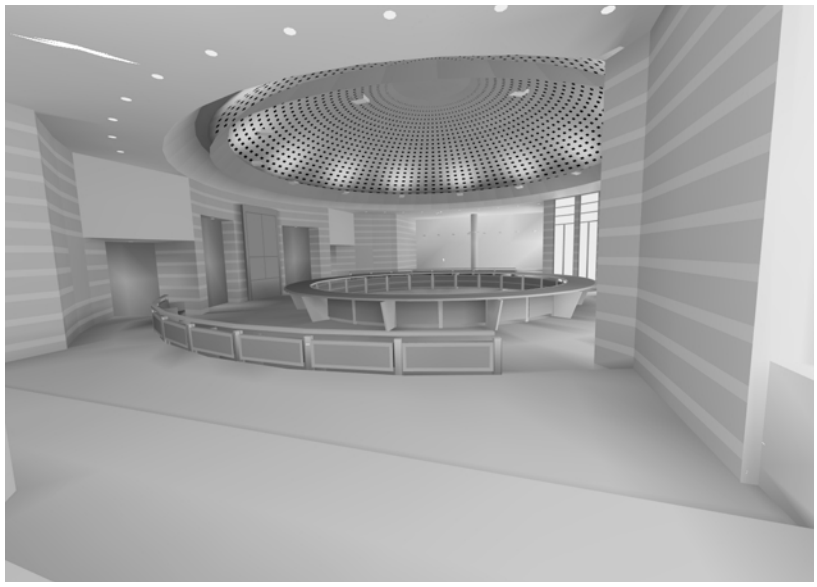

Figure 1-1 Section of the New Jerusalem City Hall

Tap the middle mouse button 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 are 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; when you are 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 do not worry—at this point you will bounce off 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 are tuning a database, because it is

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 **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. The GUI is part of `libpfutil`. See the following sample program:

`/usr/share/Performer/src/pguide/libpfutil/utilui.c`
(IRIX and Linux)
`%PFROOT%\Src/pguide\libpfutil\utilui.c`
(Windows)

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 OpenGL Performer's use of culling to ignore objects that are completely outside the field of vision. For more information about culling, see Chapter 15, "Optimizing Performance." For more detailed information on the statistics panels, see Chapter 18, "Statistics" in the *OpenGL Performer Programmer's Guide*.

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 that if you have closed the control panel, the `F1` key restores it.)

Other Motion Models

So far you have been driving. There are other default motion models provided through the `libpfui` library. These motion models can be subclassed to create your own models. On IRIX and Linux systems, you can find the source code for these motion models in `/usr/share/performer/src/lib/libpfui/`. On Windows systems, the source code is found in `%PFROOT%\Src\lib\lib\libpfui\`.

Flying

The Fly motion model provides an alternative to the Drive model. This model allows full motion in three dimensions (unlike the Drive model, which does not 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 will continue to accelerate.

You turn by 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 or 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 **Fly** button 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. Dragging up and down causes rotation about the horizontal axis parallel to the screen; 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. For instance, you can 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.

The Use of Instances

The bench objects in the City Hall scene were designed using the database concept known as instancing. For example, 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 the *OpenGL Performer Programmer's Guide* for information on this topic.

Loading Databases into OpenGL Performer

Databases do not need to be converted to a standard file format before being read to an OpenGL Performer application. Rather, unique file readers are constructed for each format to be used. OpenGL Performer can thus work with data from multiple sources concurrently, using a common software interface, without needing intermediate translation or conversion steps.

The `libpfdb` library is a collection of independent loaders, each of which loads a particular file format into OpenGL Performer. Among the loaders included in the distribution are loaders for Optimizer, Inventor, VRML, OpenFlight, Designer's Workbench, Medit, and Wavefront. Each of the `libpfdb` loaders is located in its own source directory. Users can call the `libpfdu` function **pfdLoadFile(),** which uses the extension part of the filename to determine the file format and automatically invoke the proper loader from `libpfdb`. For example, to use the visual interface from the Perfly demo with your own databases, simply list your databases after the `perfly` command

line. The `perfly` command examines the extension part of each filename to determine what format the file is in and does a run-time system lookup for a loader for that format.

Note: Many of the database loaders are contributed by companies that are OpenGL Performer developers. In particular, the OpenFlight loader is provided by Multigen, the Designer's Workbench loader is provided by Centric Software, Inc., and the VRML 2.0 loader is provided by OpenWorlds, Inc.

You can write custom loaders for whatever formats you require in your applications.This is not a difficult task, but it does require that you understand most of OpenGL Performer. It should not be undertaken until you have completed reading this book. New loaders can be written and added at any time, and the run-time lookup mechanism can find the new loader when the new file type is encountered. To see the source code for provided loaders and more information on the companies that contributed them, look in the directories under `/usr/share/Performer/src/lib/libpfdb` on IRIX and Linux under `%PFROOT\Src\lib\libpfdb` on Windows systems.

For more information about database formats, see the section "Geometry Builder Library (libpfdu)" on page 18.

Going Beyond Visual Simulation

In the Perfly demo, 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 OpenGL Performer helps you create: the visual part. In other words, what you see when you look out the window.

But there is 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 is no nonvisual sensory input at all; the user simply watches scenery move past. OpenGL 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 OpenGL Performer, the result can be a complete sensory environment, both visual and physical—creating a convincing simulation of reality. Since OpenGL 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 OpenGL Performer that lies below the surface of the demos is its ability to accelerate graphics to top-rated performance levels on SGI hardware. This means that OpenGL Performer puts a virtual SGI 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.

OpenGL Performer Basics

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

OpenGL Performer Applications

OpenGL Performer can be used in various ways. You can use it as a complete database processing and rendering system for applications such as flight simulation, driver training, or virtual reality. You can also use it in conjunction with layered application-development tools to perform the low-level portion of visual simulation development projects. In short, applications can use part or all of the features provided by OpenGL 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 OpenGL Performer in several ways. If time is short and the bottleneck is in the rendering code, OpenGL Performer's `libpr` rapid- rendering 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 OpenGL Performer's run-time format and gain the extra features that the full library, `libpf,` provides.

Library Structure of OpenGL Performer

OpenGL Performer is an extensible software toolkit for creating real-time 3D graphics. On IRIX and Linux systems, the main components of the toolkit are six libraries, typically used in their *dynamic shared object* (DSO) form with the `.so` suffix, as shown in Table 2-1; support files for those libraries (such as the header files); and source code for sample applications. On Windows systems, the DSO equivalent is a dynamic link library (DLL) with a corresponding file suffix of `.dll`.

Table 2-1 OpenGL Performer Libraries

DSO/DLL Name	Header File	Description
`libpf.so` `libpf.dll`	`pf.h`	Main OpenGL 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 that greatly facilitate the construction of database loaders and converters. Tools include a sophisticated triangle mesher and state sharing for high-performance databases.
`libpfutil.so` `libpfdu-util.dll`	`pfutil.h`	Utility functions library. Note that `libpfdu-util.dll` is a combination of `libpfdu` and `libpfutil`.
`libpfui.so` `libpfui.dll`	`pfui.h`	User interface library.
`libpfv.so` `libpfv.dll`	`pfv.h`	A graphical viewer library that provides for the easy construction of applications.
`libpfmpk.so` `libpfmpk.dll`	`pfmpk.h`	Library for importing display-configuration information from files using the OpenGL Multipipe SDK configuration file format.
`libpfdb`	`pfdb.h`	Collection of libraries containing the load, convert, and store routines for numerous file formats.

Note: Throughout this guide, a reference to DSO files will pertain to both DSO and DLL files unless otherwise noted.

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, procedure call performance, and caching behavior. However, `libpr` still provides an implementation and portability abstraction layer that simplifies the following discussions.

In addition to the core libraries, OpenGL 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 OpenGL 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 OpenGL Performer libraries and the operating system software.

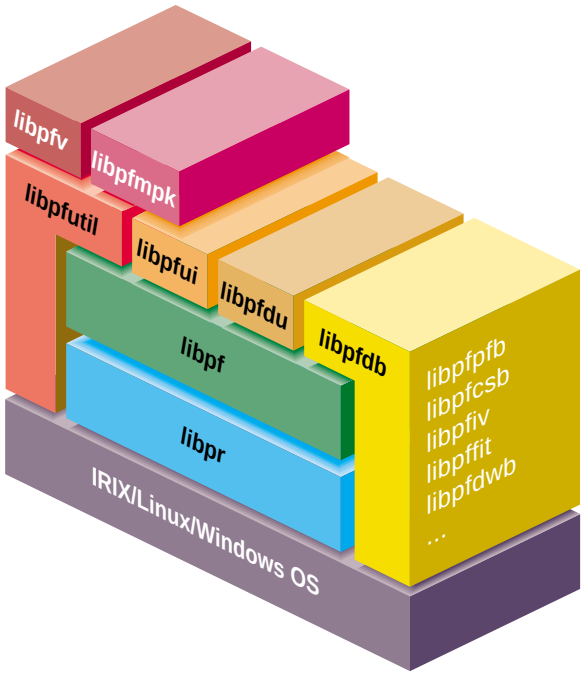

Figure 2-1 OpenGL Performer Library Hierarchy

All OpenGL Performer features are provided as a layer above the operating system and the graphics library. However, OpenGL Performer does not isolate application programs from the operating system or the graphics library, however. Even when using OpenGL Performer to its fullest extent, applications have direct and free access to all system

layers—including not only `libpf`, `libpr`, `libpfdu` ,`libpfutil`, `libpfui`, `libpfv`, `libpfmpk`, and the `libpfdb` loader, but also the OpenGL graphics library and the operating system. You are free to choose which of the libraries best suits your needs. You may want to build your own toolkits on top of `libpr` (but you still link with `libpf`; you just do not use any `libpf` features), or you can take advantage of the visual simulation development environment that `libpf` provides.

OpenGL Performer defines a run-time-only database through its programming interface; it does not define an archival database or file format. Applications import their databases into OpenGL 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 30 popular database formats and how to export scene graphs in the open Designer's Workbench and Medit formats (see *OpenGL Performer Programmer's Guide* for more information).

Library Features

This section lists the features of the OpenGL Performer libraries. An application can use all or just part of the features. You can use these features in conjunction with or extend them with other application development tools.

High-Performance Rendering Library (`libpr`)

`libpr` consists of many facilities generally required in most visual simulation and real-time graphics applications, such as:

- 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

- Windowing and video channel management utilities
- Asynchronous filesystem I/O
- Shared memory allocation facilities
- High-resolution clocks and video-interval counters

Visual Simulation Application Library (`libpf`)

- Multiple graphics pipeline capability
- Multiple windows per graphics pipeline
- Multiple display channels and video channels per window
- Hierarchical scene graph construction and real-time editing
- Multiprocessing (parallel simulation, intersection, cull, draw processes, and asynchronous database management)
- System stress and load management
- Asynchronous database paging
- Morphing
- Level-of-detail model switching, with fading or morphing
- Rapid culling to the viewing frustum
- Intersections and database queries
- Dynamic and static coordinate systems
- Fixed-frame-rate capability
- Shadows and spotlights
- Visual simulation features
  - Environmental model
  - Light points, both raster and calligraphic
  - Animation sequences
  - Sophisticated fog and haze control
  - Landing light capabilities
  - Billboarded geometry

Geometry Builder Library (`libpfdu`)

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

Utility Library (`libpfutil`)

- Processor isolation routines
- GLX mixed mode utilities
- Device input and event handling
- 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`)

- Motion models, including trackball, fly, and drive
- Collision models

A Graphical Viewer Library (`libpfv`)

- Reading and writing XML files
- Specifying complex display configuration (pipes, windows, and channels) from a file or through API calls
- Tracking mouse and keyboard input
- Setting up user interaction with 3D scene elements
- Managing multiple scene graphs (worlds)

- Managing multiple camera positions (views)
- Extending program functionality using program modules

A Configuration-Import Library (`libpfmpk`)

- Imports display-configuration information from files using the OpenGL Multipipe SDK configuration file format.
- Configures OpenGL Performer pipes, windows, and channels according to the configuration file specifications.

Database Loader Library (`libpfdb`)

- Common software interface to read files
- Supports a wide variety of file formats.
- Source code included as templates for customization

Overview of the OpenGL Performer Library Structure

This section outlines the basic elements of each library.

`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 critical path pipeline stages are:

- APP
- CULL
- DRAW

The application (APP) 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 to tailor the application to the number of available CPUs, as shown in Figure 2-2.

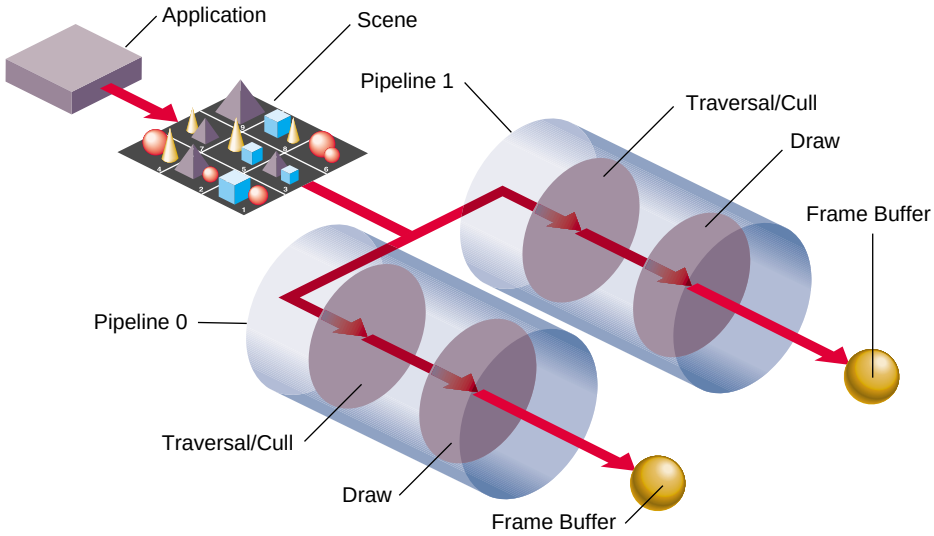

Figure 2-2 Parallel Pipeline Processes

An application might have multiple rendering pipelines drawing to multiple graphics pipelines with separate processes. The CULL task of the rendering pipeline can itself be multithreaded.

OpenGL Performer provides additional, asynchronous stages for various computations:

- INTERSECTION—intersects line segments with the database for things like collision detection and line-of-sight determination, and may be multithreaded.
- COMPUTE—for general, asynchronous computations.
- DATABASE—for asynchronously loading files and adding to or deleting files from the scene graph.

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

For more information about multiprocessing stages, see Chapter 11, “Multiprocessing.”

Display

`libpf` provides software constructs to facilitate visual database rendering. 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.

Frame Control

OpenGL Performer is designed to run at a fixed frame rate specified by the application. OpenGL 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.

OpenGL 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 and between the host application and the graphics subsystem is provided. This also supports simulations that display multiple simultaneous views on different hardware displays.

Visual Database (pfScene)

A visual database is a graph of nodes with a pfScene node as its root. 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. OpenGL Performer supplies sample source code that shows how to construct a scene from several popular database formats; see *OpenGL Performer Programmer's Guide* for more information.

`Qlibpf` supports a general database scene graph hierarchy, defined as a directed acyclic graph of nodes. OpenGL 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

Geometry and leaf nodes are the following:

- pfGeode—Geometry node
- pfBillboard—Geometry that rotates to face the viewpoint
- pfText—Geometry based upon pfFont and pfString
- pfASD—Active Surface Definition for morphing geometry and continuous level of detail (LOD) measurement
- pfLightSource—User-manipulatable lights that support high-quality spotlights and shadows

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

- Culling the scene to the visible portion in the viewing frustum.
- Comprehensive, user-directed database 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 using callbacks.

Special Features (pfEarthSky, pfSequence, pfASD)

`libpf` provides an environmental model called a pfEarthSky, consisting of ground and sky polygons, which 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 are not 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.

Active Surface Definition (pfASD) is a library that handles real-time surface meshing and blending in a multiprocessing and multichannel environment. The pfASD approach uses a modeling terrain that is a single, connected surface rather than a collection of patches.

A pfASD surface contains several hierarchical level of detail (LOD) meshes where one level encapsulates a coarser level of detail than the next. When rendering a pfASD surface, an evaluation function selects polygons from the appropriate LODs, and constructs a valid meshing to best approximate a real terrain. An evaluation function, for example, might be based on distance.

Unlike existing LOD schemes, pfASD selects triangles from many different LODs and combines them into a final surface that transitions smoothly between LODs without cracks. This feature lets a fly-through over a surface use polygons from higher LODs for drawing nearby portions of the surface in combination with polygons from low LODs that represent distant portions of the surface.

`libpr`—High-Performance Rendering Library

`libpr` is a low-level graphics library supporting various functions useful for any high-performance graphics application.

High-Performance Geometry Rendering

Many graphics applications are limited in sending graphics commands to the Geometry Pipeline by CPU overhead. A pfGeoSet (or pfGeoArray) 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

OpenGL Performer optimizes graphics library performance by managing state changes, and provides functions to control aspects of the 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.

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

Graphics State Encapsulation

A pfState is an encapsulation of graphics that renders lighting, texturing, and fog—the state settings for a graphics context. Loading a pfState ensures that the graphics pipeline is configured appropriately, regardless of previous graphics state. pfGeoStates describe the state of the geometry in pfGeoSets or pfGeoArrays, and are used for simplifying and accelerating graphics state management.

Display Lists

OpenGL Performer supports special `libpr` display lists. They do not use graphics library objects, but rather a simple token/data mechanism that does not 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 display lists in OpenGL 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)

OpenGL Performer supports global color tables that can define the colors used by pfGeoSets and pfGeoArrays. You can use color tables for special effects such as infrared lighting, and you can switch them in real time.

Light Points

Light points, defined by the pfLPointState state object, can simulate highly emissive objects such as runway lights, approach lights, strobes, beacons, and street lights. The size, direction, shape, color, and intensity of these lights can be controlled.

Calligraphic extensions to pfLPointState provide a means of displaying exceptionally bright light points on non-raster display systems.

For more information about pfLPointState, see Chapter 16, "Light Points," in the *OpenGL Performer Programmer's Guide*.

pfObjects

OpenGL Performer is an object-oriented API. Basic object function, such as creation, deletion, and printing, are inherited from pfObject. Basic memory management is done through pfMemory.

Asynchronous File I/O (pfFile)

A simple nonblocking file access method is provided to allow applications to retrieve file data during real-time operation.

Memory Allocation (pfDataPool)

OpenGL Performer includes routines to allocate memory from the application process *heap* or from *shared memory arenas*. Shared memory 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,)

OpenGL 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.

The pfWindow Windowing Functions

OpenGL Performer provides window-system-independent window routines to allow greater portability of applications. For information about these window routines, see Chapter 11, "Windows," in the *OpenGL Performer Programmer's Guide*.

For sample programs involving windows and input handling on IRIX and Linux systems, see the following directories:

```
/usr/share/Performer/src/pguide/{libpr,libpf,libpfutil,libpfui}
```

On Windows systems, see these directories:

```
%PFROOT%\Src\pguide\{libpr,libpf,libpfutil,libpfui}
```

libpfdu—Geometry Builder Library

Although OpenGL Performer does not define a file format, it does provide sample source code for importing numerous other database formats into OpenGL Performer's run-time structures. Figure 2-3 shows how databases are imported into OpenGL Performer: first, a user creates a database with a modeling program, and then an OpenGL Performer-based application imports that database using one of the many importing routines.

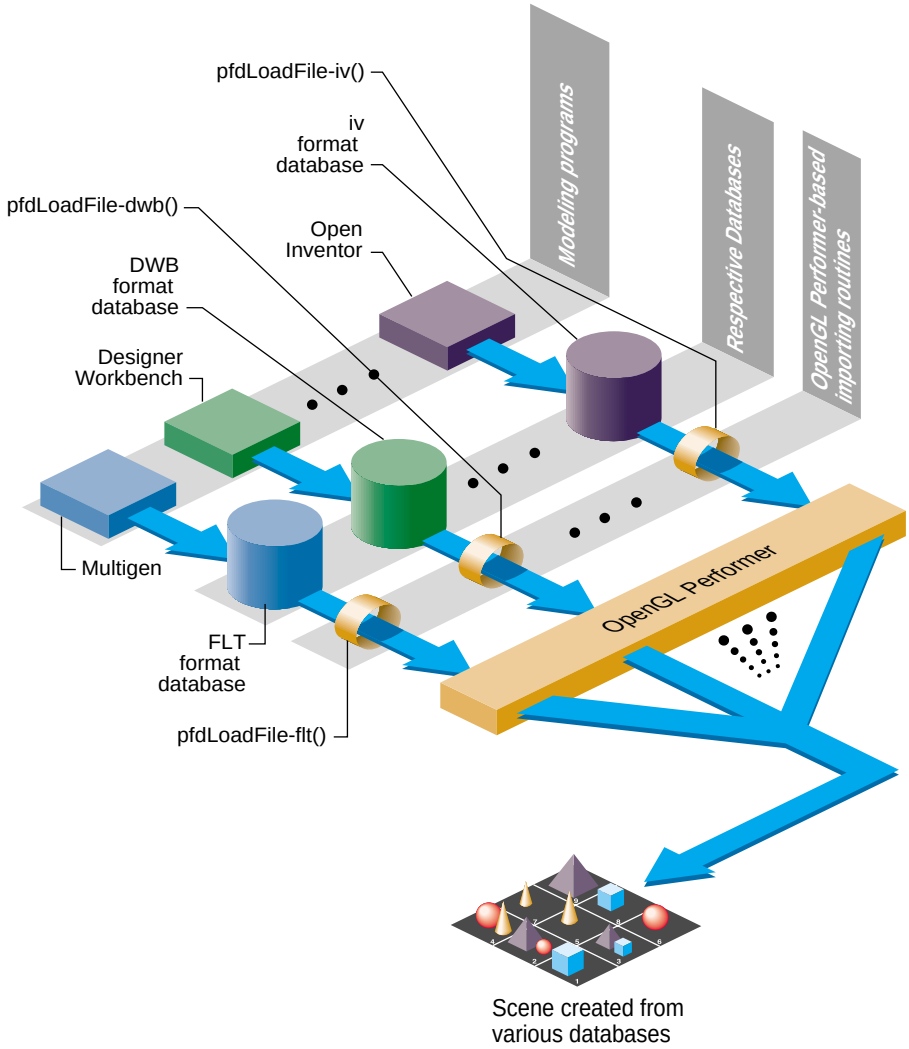

Figure 2-3 Relationship of OpenGL Performer to Database Formats

OpenGL 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.

`libpfdu` is a database utilities library that provides helpful functions for constructing optimized OpenGL Performer data structures and scene graphs. It is mainly used by database loaders, which take an external file format containing 3D geometry and graphics state and load them into OpenGL 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. However, these data structures are often mutually exclusive with effective OpenGL Performer run-time structures.

Database Builder

`libpfdu` contains many utilities, including DSO support for database loaders and their modes, and file path support. The heart of `libpfdu` is the OpenGL 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.

Geometric primitives with their corresponding graphics state are sent one at a time to the builder. When the builder has received all the data, the builder can return optimized OpenGL Performer data structures, which can be used as a part of a scene graph. The builder hashes geometry into different bins, based on the attribute binding types and associated graphics state of the geometry. The builder also keeps track of graphics state elements, such as textures, materials, light models, and fog, and shares state elements whenever possible.

The builder creates pfGeoSets that contain triangle meshes created by running the original geometry through the `libpfdu` triangle-meshing utility.

For each pfGeoSet, the builder creates a pfGeoState (OpenGL 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 places them in a leaf node (pfGeode), and optionally creates 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: The builder allows the user to extend the notion of a graphics state by registering callback functionality through the builder API, and then treating this state or functionality like any other OpenGL Performer state or mode (although such uses of the builder are slightly more complicated).

`libpfv`—A Graphical Viewer Library

The library `libpfv` provides for the easy construction of modular, interactive OpenGL Performer applications.

The library `libpfv` supports the following features:

- Reading and writing XML files
- Specifying complex display configuration (pipes, windows, and channels) from a file or through API calls
- Tracking mouse and keyboard input
- Setting up user interaction with 3D scene elements
- Managing multiple scene graphs (worlds)
- Managing multiple camera positions (views)
- Extending program functionality using program modules

The principal class in `libpfv` is the pfvViewer. It allows complex multiworld and multiview applications to be implemented in a modular fashion, allowing individual features to be encapsulated into configurable and re-usable modules.

In addition to `libpfv`, OpenGL Performer includes ready-to-use modules that provide the following features:

- Loading geometry into a pfvViewer world
- Picking geometry under the mouse pointer
- Manipulating geometry (rotating, translating, scaling, deleting)
- Navigating through a world using mouse and keyboard controls
- Controlling the render style of models
- Setting up colorful earth and sky backgrounds

- Displaying 2D images in overlay
- Saving snapshots of the rendered images
- Smoothly transitioning from one world to another
- Collecting and displaying statistics

`libpfmpk`—A Configuration-Import Library

A typical OpenGL Performer program starts with defining how many pipes, windows, and channels it requires. This code has to change every time you target the application at a new hardware configuration. The software product OpenGL Multipipe SDK solves this problem by providing a file format for specifying different display configurations. Loading such a configuration file determines the pipe/window/channel configuration that the program uses.

The library `libpfmpk` facilitates importing configuration files that use the OpenGL Multipipe SDK configuration file format. This library configures the OpenGL Performer application using the configuration file specifications. As a result, display configuration becomes easier and quicker to change.

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 functions that OpenGL Performer provides, you do not need to know X or IRIS IM to use windows. However, you might want to integrate pfWindows with a Motif application or have a pfWindow use a designated Motif window.

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 does not have window or event routines. The *OpenGL Porting Guide* provides more information on porting from IRIS GL to OpenGL, and the sample applications distributed with OpenGL Performer provide many examples of programs that compile and run with either IRIS GL or OpenGL.

Most of the differences between IRIS GL and OpenGL are transparent to a developer using OpenGL Performer. The most significant difference between IRIS GL and OpenGL is how Performer handles windows and input. These differences are covered by pfWindows that provide a GL-independent windowing layer. Graphics rendering and state calls made through the OpenGL Performer API are also GL-independent.

You will notice differences between IRIS GL and OpenGL when direct GL calls are used outside of the OpenGL Performer interface. There are relatively few circumstances in which your OpenGL Performer-based program needs to call graphics library routines directly. Making outside calls usually happens only in DRAW callbacks. For more information, see "Customizing OpenGL Performer Traversals" on page 106.

For information on compiling and linking OpenGL Performer applications, see the *OpenGL Performer Programmer's Guide*.

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]

Visual simulation systems have advanced significantly due to advances in hardware and software, and to a greater understanding of human perceptions. For example, the Mars Sojourner Rover, the land rover on Mars, was simulated by researchers with an OpenGL Performer application.

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 actual latency and increasing the 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

[1] In recognition of the ingenuity of this system, OpenGL 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.

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 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 a 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 user input. 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. However, it is difficult to maintain 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 is 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 the potential visibility of each object during the culling process by considering other objects within the scene that may occlude 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 Chapter 15, "Optimizing Performance."

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 Chapter 15, "Optimizing Performance."

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. The behavior can 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 the section, “pfSequence Nodes” in the *OpenGL Performer Programmer’s Guide*.

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 because 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 the area of a pixel rather than simply be a sample taken at the center of the pixel. This idea is easily stated but difficult to implement while maintaining high performance.

InfiniteReality continues the *RealityEngine* antialiasing approach known as 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 *subpixel Z-buffer*) per image pixel allows opaque polygons to be drawn in an arbitrary order because 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 the 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, which serve 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. With the InfiniteReality graphics subsystem, you can have very large textures, called *cliptextures*, (up to 8Mx8M texels).

For more information about texture mapping and cliptextures, see Chapter 8, "Geometry," and Chapter 10, "ClipTextures," in the *OpenGL Performer Programmer's Guide*.

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 "Rendering Slices of Shapes" on page 212). For example, to display a tree using textures and billboards, you would create a texture map of a tree (from a photograph, perhaps), marking the background (any part of the texture that 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 surface normal of the geometry to compute the index of each screen pixel 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 with systems in which 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 requirements and techniques.

Morphing

The shared-memory, multiprocessed system architecture with high bandwidth for graphics subsystems in the SGI product line provide ideal systems for real-time high quality character animation. Vertex position, colors, normal vectors, and texture coordinates can all be interpolated between two versions of a model, a process known as *morphing*, with the OpenGL Performer pfEngine and pfFlux objects. You can also apply more complex functions between multiple versions of a model. You can use morphing 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).

For more information about morphing, see Chapter 14, "Dynamic Data", in the *OpenGL Performer Programmer's Guide*.

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.

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. You can often improve the quality of the interpolated result 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*. With 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 3D 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 together 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 and example databases that are well integrated with OpenGL 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 explore.

PART TWO

Programming with OpenGL Performer II

Chapter 3, “OpenGL Performer Programming Interface.”

Chapter 4, “Introduction to OpenGL Performer Concepts.”

Chapter 5, “Creating a Display with pfChannel.”

Chapter 6, “Creating Scene Graphs.”

Chapter 7, “Creating Geometry with pfGeoSets.”

Chapter 8, “Specifying the Appearance of Geometry with pfState and pfGeoState.”

Chapter 9, “Placing Geometry in a Scene.”

Chapter 10, “Controlling Frame Rate.”

Chapter 11, “Multiprocessing.”

Chapter 12, “Database Paging.”

Chapter 13, “Intersection Testing.”

Chapter 14, “Creating a User Interface.”

Chapter 15, “Optimizing Performance.”

OpenGL Performer Programming Interface

This chapter describes the fundamental ideas behind the OpenGL Performer programming interface in the following sections:

General Naming Conventions

The OpenGL Performer API uses naming conventions to help you understand what a given command will do and even predict the appropriate names of routines for desired functionality. 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.

The API is largely object-oriented; it contains classes of objects comprised of methods that do the following:

- Configure their parent objects.
- Apply associated operations, based on the current configuration of the object.

Both C and C++ bindings are provided for OpenGL Performer. In addition, naming conventions provide a consistent and predictable API and indicate the kind of operations performed by a given command.

Prefixes

The prefix of the command tells you in which library a C command or C++ class is found. All exposed OpenGL Performer C commands and C++ classes begin with "pf". The utility libraries use an additional prefix letter, such as "pfu" for the `libpfutil` general utility library, "pfi" for the `libpfui` input handling library, and "pfd" for the `libpfdu`

database utility library. `Libpr` level commands still have the `pf' prefix as they are still in the main `libpf` library.

Header Files

Each OpenGL Performer library contains a main header file in `/usr/include/Performer` that contains type and class definitions, the C API for that library, and global routines that are part of the C and C++ API. `Libpf` is broken into two distinct pieces: the low-level rendering layer, `libpr`, and the application layer, `libpf`, and each has their own main header file: `pr.h` and `pf.h`. Because `libpf` is considered to include `libpr`, `pf.h` includes `pr.h`. C++ class header files are found under `/usr/include/Performer/`{pf, pr, ...} on IRIX and Linux systems. On Windows, the header files are under `%PFROOT%\Include\Performer`. Each class has its own C++ header file and that header must be included to use that class.

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

#include <Performer/pf/pfGroup.h>
.....
pfGroup *group;
```

Naming in C and C++

All C++ class method names have an expanded C counterpart. Typically, the C routine will include the class name in the routine, whereas the C++ method will not.

```
C: pfGetPipeScreen();
C++: pipe->getScreen();
```

For some very general routines on the most abstract classes, the class name is omitted. This is the case with the child API on pfNodes:

```
C: pfAddChild(node,child);
C++: node->addChild(child);
```

Command and type names are mixed case; the first letter of a new word in a name is capitalized. C++ method names always start with a lowercase letter.

```
pfTexture *texture;
texture->loadFile();
```

Abbreviations

Type names do not use abbreviations. The C API acting on that type will often use abbreviations for the type names, as does the associated tokens and enums.

In procedure names, a name is always abbreviated or never abbreviated, and the same abbreviation is always be used and is in the pfNew* C command. For example, the pfTexture object uses "Tex" in its API, such as **pfNewTex()**. If a type name has multiple words, the abbreviation uses the first letter of the first words and then the first syllable of the last word.

```
pfPipeWindow *pwin = pfNewPWin();
pfPipeVideoChannel *pvchan = pfNewPVChan();
pfTexLOD *tlod = pfNewTLOD();
```

Macros, Tokens, and Enums

Macros, tokens, and enums all use full uppercase. Token names associated with a class and methods of a class start with the abbreviated name for that class, such as texture to “tex” in PFTEX_SHARPEN.

Class API

The API of a given class, such as pfTexture, is comprised of:

- API to create an instance of the object
- API to set parameters on the object
- API to get those parameter settings
- API to perform actions on the configured object

Object Creation

Objects are always created with:

```
C: pfThing *thing = pfNewThing();
C++: pfThing *thing = new pfThing;
```

`Libpf` objects are automatically created out of the shared memory arena. Libpr objects take as an argument an arena pointer that, if NULL, causes allocation off the heap.

Set Routines

A set routine has the form:

```
C: pfThingParam(thing, ... ) (note no ‘Set’ in the name)
C++: thing->setParam()
```

Set routines are usually very fast and are not order dependent. Work required to process the settings happens once when the object is first used after settings have changed. If particularly expensive options must be done, there will be a pfConfigThing routine or method to explicitly force this work that must be called before the object is to be used.

Get Routines

For every "set" there is a matching "get" routine to get back the value that was set.

```
C: pfGetThingParam(thing, ... )
C++: thing->getParam()
```

If the set/get is for a single value, that value is usually the return value of the routine. If there are multiple values together, the "get" routine will use pointers to result variables.

Getting Current In-Use Values

Get routines return values that have been previously set by the user, or default values if no settings have been made. Sometimes a value other than the user-specified value is currently in use and that is the value that you would like to get. For these cases, there is a separate "getCur" routine to get the current value in use.

```
C:   pfGetCurThingParam()
C++: thing->getcurParam()
```

These "cur" routines may only be able to give reasonable values in the process in which associated operations are happening. For example, to get the current texture (**pfGetCurTex()**), you need to be in the DRAW process, because that is the only process that has a current texture.

Action Routines

An action routine has the following form:

```
C: pfVerbThing(), such as pfApplyTex()
C++: thing->verb(), such as tex->apply()
```

Action routines can have parameter scope and apply only to that parameter. These routines have the following forms:

```
C: pfVerbThingParam(), such as pfApplyTexMinLOD()
C++: thing->verbParam(), such as tex->applyMinLOD()
```

The APPLY and DRAW action routines do graphics operations and so must happen either in the DRAW process or in display list mode.

```
C: pfApplyGState()
pfDrawGSet()
C++: gstate->apply()
gset->draw()
```

Enable and Disable of Modes

You can enable land disable with **pfEnable()** and **pfDisable()**, respectively.

pfEnable() and **pfDisable()** take PFEN_* tokens, naming the graphics state operation to enable or disable. **pfGetEnable()** is used to query enable status, and will return 1 or 0 if the given mode is enabled or disabled, respectively.

```
ex: pfEnable(PFEN_TEXTURE), pfDisable(PFEN_TEXTURE),
pfGetEnable(PFEN_TEXTURE);
```

Mode, Attribute, or Value

Class instances are configured by having their internal fields set. These fields may be simple modes or complex attribute structures. Mode values are ints or tokens, attributes are typically pointers to objects, and values are floats.

```
pfGStateMode(gstate, PFSTATE_DECAL, PFDECAL_LAYER)
pfGStateAttr(gstate, PFSTATE_TEXTURE, texPtr)
pfGStateVal(gstate, PFSTATE_ALPHAREF, 0.5)
```

Base Classes

OpenGL Performer provides an object-oriented programming interface to most of its data structures. Only OpenGL Performer functions can change the values of elements of these data structures; for example, you must call **pfMtlColor()** to set the color of a pfMaterial structure rather than modifying the structure directly.

For a more transparent type of memory, OpenGL Performer provides pfMemory. All object classes are derived from pfMemory. pfMemory instances must be explicitly allocated with the new operator and cannot be allocated statically, on the stack, or included directly in other object definitions. pfMemory is managed memory; it includes special fields, such as size, arena, and ref count, that are initialized by the pfMemory **new()** function.

Some very simple and unmanaged data types are not encapsulated for speed and easy access. Examples include pfMatrix, pfSphere and pfVec3. These data types are referred to as public structures and are inherited from pfStruct.

Unlike pfMemory, pfStructs can be:

- Allocated statically.
- Allocated on the stack.
- Included directly in other structure and object definitions.

pfStructs allocated off the stack or allocated statically are not in the shared memory arena and thus are not safe for multiprocessed use. Also, pfStructs allocated off the stack in a procedure do not exist after the procedure exits so they should not be given to persistent objects, such as a pfVec3 array of vertices for a pfGeoSet.

To allow some functions to apply to multiple data types, OpenGL 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 that 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 the following appends *child* to the list of children belonging to *dcs*.

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

Because the C language does not directly express the notion of classes and inheritance, arguments to functions must be cast before being passed, for example:

```
pfAddChild((pfGroup*)dcs, (pfNode*)child);
```

In the example above, no such casting is required because OpenGL Performer provides macros that perform the casting when compiling with ANSI C, for example:

```
#define pfAddChild(g, c) pfAddchild((pfGroup*)g, (pfNode*)c)
```

Note: Using automatic casting eliminates type checking—the macros 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. 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 3-1. OpenGL Performer does not use multiple inheritance, so each class has only one parent in the graph.

Note: It is 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.

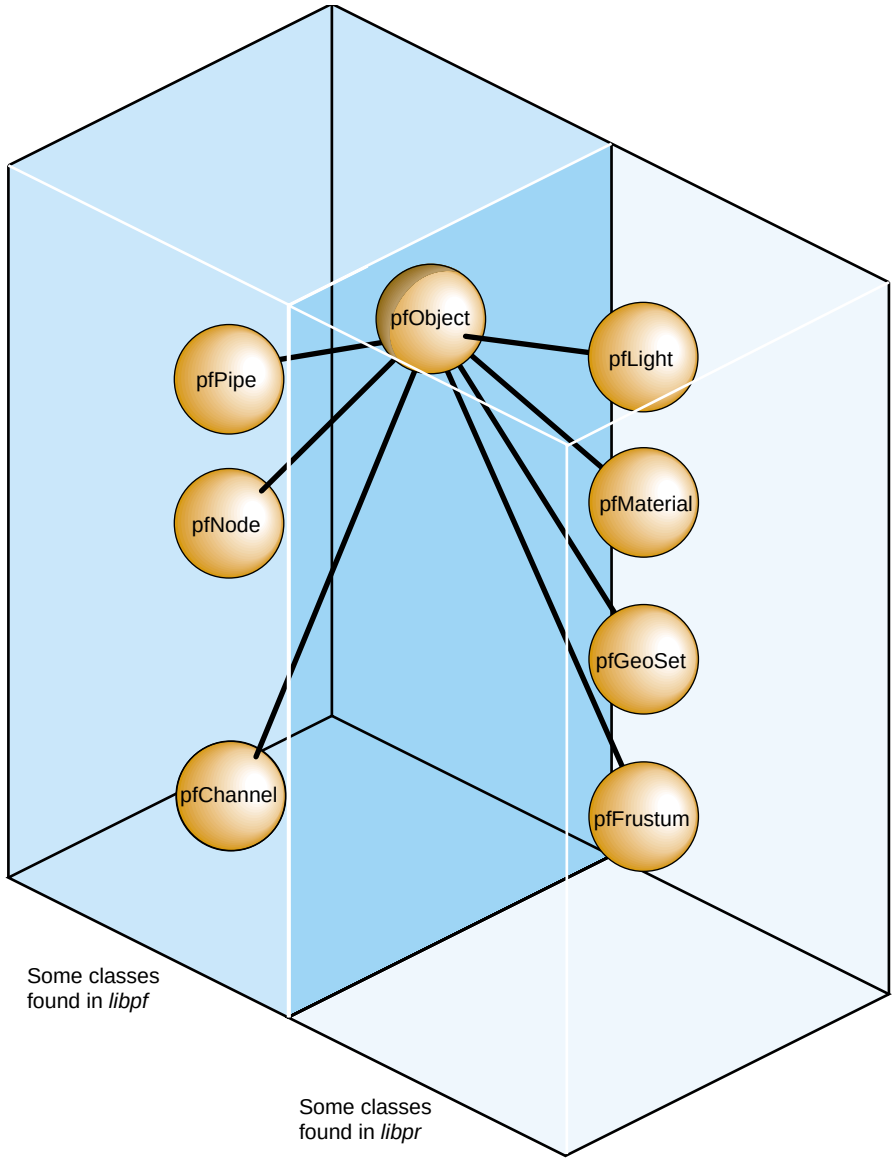

Figure 3-1 Partial Inheritance Graph of OpenGL Performer Data Types

OpenGL 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 OpenGL 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, a pfPipeWindow has a pfWindow, and a pfPipeVideoChannel has a pfVideoChannel. In these cases, the first object in each pair provides functions corresponding to those of the second. For example, pfFrustum has the following routine:

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

and pfChannel has a corresponding routine:

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

`Libpr` and `Libpf` Objects

All of the major classes in OpenGL Performer are derived from the pfObject class. This common, base class unifies the data types by providing common attributes and functions. `Libpf` objects are further derived from pfUpdatable. The pfUpdatable abstract class provides support for automatic multibuffering for multiprocessing. pfObjects have no special support for multiprocessing and so all processes share the same copy of the pfObject in the shared arena. `libpr` objects allocated from the heap are only visible in the process in which they are created or in child processes created after the object. Changes made to such an object in one process are not visible in any other process.

Explicit multibuffering of pfObjects is available through the pfFlux class. In general, `libpr` provides lightweight and low-level modular pieces of functionality that are then enhanced by more powerful `libpf` objects.

User Data

The primary attribute defined by the pfObject class is the custom data a user can define on any pfObject called “user data.” pfUserData attaches the user-supplied data pointer to the user data attribute. pfUserDataSlot attaches the user supplied data pointer to the given user data slot. Example 3-1 shows how to use user data.

Example 3-1 How to Use User Data

```
typedef struct
    {
      float coeffFriction;
      float density;
      float *dataPoints;
    }
    myMaterial;

    myMaterial     *granite;

    granite = (myMaterial *)pfMalloc(sizeof(myMaterial), NULL);
    granite->coeffFriction = 0.5f;
    granite->density = 3.0f;
    granite->dataPoints = (float *)pfMalloc(sizeof(float)*8, NULL);
    graniteMtl = pfNewMtl(NULL);

    pfUserData(graniteMtl, granite);
```

pfDelete() and Reference Counting

You can place types of data objects in OpenGL Performer can be placed in a hierarchical scene graph, using instancing (see *OpenGL Performer Programmer's Guide*) when an object is referenced multiple times. Scene graphs can become quite complex, which can cause problems if you are 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.

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 OpenGL Performer routines shown in Table 3-1.

Table 3-1 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()**	Modifies 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 3-2 demonstrates how reference counts are incremented and decremented.

Example 3-2 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 by OpenGL Performer routines would normally be sufficient for your needs. However, the routines **pfRef()**, **pfUnref()**, and **pfGetRef()** allow you to increment, decrement, and retrieve the reference count of a `libpr` object if

you want to do so. (These routines also work with objects allocated by **pfMalloc()**; see the *OpenGL Performer Programmer's Guide* for more information).

You can delete an object whose reference count is equal to 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, and then it deletes the current object. **pfDelete()** does not 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 3-3 illustrates the use of **pfDelete()** with `libpr`.

Example 3-3 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 3-4 illustrates the use of **pfDelete()** with `libpf`.

Example 3-4 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 they guarantee mutual exclusion when multiprocessing.
- The counts of objects added to a pfDispList are not incremented due to performance considerations.
- In the multiprocessing environment of `libpf`, the successful deletion of a pfNode does not have immediate effect but is delayed one or more frames until all processes in all processing pipelines are done with the node. This accounts for the fact that pfDispLists do not reference-count their objects.
- **pfUnrefDelete(obj)** is shorthand for:

  ```
  if(pfUnref(obj) ==0)
      pfDelete(obj);
  ```

  This is true when pfUnrefGetRef is atomic.

- An object whose count reaches zero is not automatically deleted by OpenGL 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 3-5.

Example 3-5 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);
```

The routine **pfMalloc()** and the related routines provide a consistent method to allocate memory, either from the user's heap (using the C library **malloc()** function) or from a shared memory arena (using the IRIX **malloc()** function).

Printing Objects with pfPrint()

pfPrint() can print many different kinds of objects to a fill. For example, you can print nodes and GeoSets. To do so, you specify in the argument of the function the object to print, the level of verbosity, and the destination file. An additional argument, *which*, specifies different data according to the type of object being printed.

The different levels of verbosity include:

- PFPRINT_VB_OFF—no printing.
- PFPRINT_VB_ON—minimal printing (default).
- PFPRINT_VB_NOTICE—minimal printing (default).
- PFPRINT_VB_INFO—considerable printing.
- PFPRINT_VB_DEBUG—exhaustive printing.

If the object to print is a type of pfNode, *which* specifies whether the print traversal should only traverse the current node (PFTRAV_SELF) or the entire scene graph where the node specified in the argument is the root node (PFTRAV_SELF | PFTRAV_DESCEND). For example, to print an entire scene graph, in which *scene* is the root node, to the file *fp*, with default verbosity, use the following line of code.

```
file = fopen (“scene.out”,”w”);
pfPrint(scene, PFTRAV_SELF | PFTRAV_DESCEND, PFPRINT_VB_ON, fp);
fclose(file);
```

If the object to print is a pfFrameStats, *which* should specify a bitmask of the frame statistics classes that you want printed. The values for the bitmask include:

- PFSTATS_ON enables the specified classes.
- PFSTATS_OFF disables the specified classes.
- PFSTATS_DEFAULT sets the specified classes to their default values.
- PFSTATS_SET sets the class enable mask to enmask.

For example, to print select classes of a pfFrameStats structure, *stats*, to stderr, use the following line of code.

```
pfPrint(stats, PFSTATS_ENGFX | PFFSTATS_ENDB | PFFSTATS_ENCULL,
   PFSTATS_ON, NULL);
```

If the object to print is a pfGeoSet, *which* is ignored and information about that pfGeoSet is printed according to the verbosity indicator. The output contains the types, names, and bounding volumes of the nodes and pfGeoSets in the hierarchy. For example, to print the contents of a pfGeoSet, *gset*, to stderr, use the following line of code:

```
pfPrint(gset, NULL, PFPRINT_VB_DEBUG, NULL);
```

Note: When the last argument, *file*, is set to NULL, the object is printed to stderr.

Determining Object Type

Sometimes you have a pointer to a pfObject but you do not know what it really is—is it a pfGeoSet, a pfChannel, or something else? **pfGetType()** returns a pfType that specifies the type of a pfObject. You can use pfType to determine the class ancestry of the object. Another set of routines, one for each class, returns the pfType corresponding to that class, For example, **pfGetGroupClassType()** returns the pfType corresponding to pfGroup.

pfIsOfType() tells whether an object is derived from a specified type, as opposed to being the exact type.

With these functions you can test for class type as shown in Example 3-6.

Example 3-6 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 OpenGL 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 that use subclassing.

The pfType returned from **pfGetType()** is useful for programs but is not in a readable form for you. Calling **pfGetTypeName()** on a pfType returns a null-terminated ASCII string that identifies an object's type. For a pfDCS, for example, **pfGetTypeName()** returns the string"pfDCS." The type returned by **pfGetType()** can then be compared to a class type using **pfIsOfType()**. Class types can be returned by methods such as **pfGetGroupClassType()**.

Introduction to OpenGL Performer Concepts

This chapter describes the basic classes that implement the database- to- display pipeline in the following sections:

Scene-to-Screen Path

A description of your world is encapsulated in the scene graph, and a view into the world is described by a pfChannel. This view is rendered by an OpenGL Performer software pipeline, pfPipe, into a window, pfPipeWindow, on a selected screen. This path of operation and associated classes is shown in Figure 4-1.

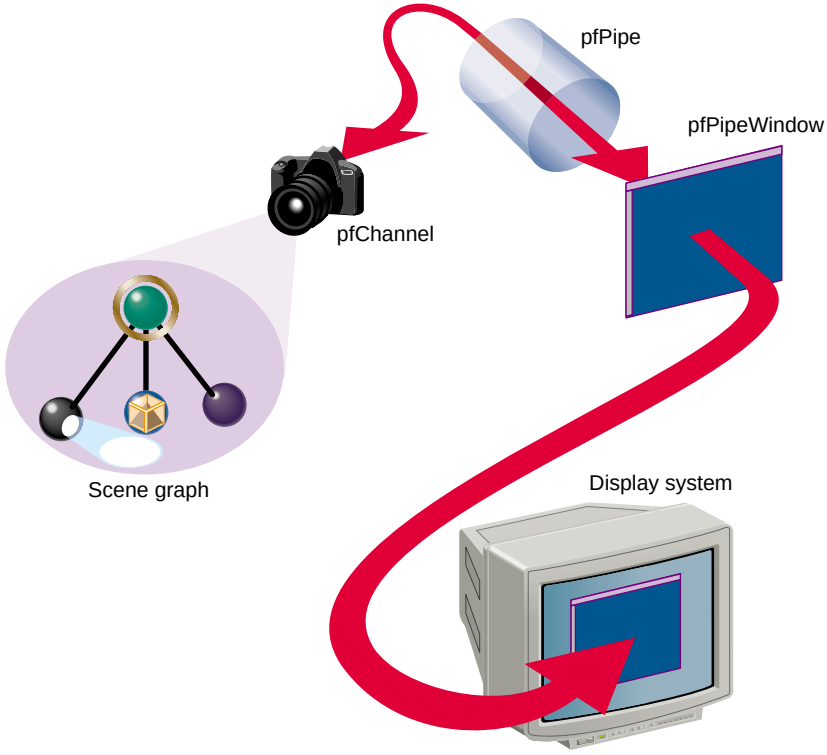

Figure 4-1 Data-to-Display

Scene Graph

A scene graph is a directional, acyclic graph (DAG). Its structure determines the order of operation of its data.

pfNode is the base class of all scene graph node types. A node might hold, for example, the data for a geometry. Different node types provide mechanisms for grouping, animation, level of detail, and other concepts that are applied when the scene graph is traversed with a traverser.

Scene Graph Hierarchy

The nodes in a scene graph are arranged in a hierarchy, as shown in Figure 4-2.

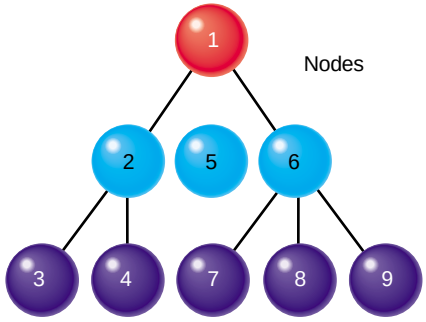

Figure 4-2 Scene Graph Hierarchy

The hierarchy of nodes can have many meanings, for example:

- The child nodes may be part of the parent node. For example, the parent node might encapsulate a light bulb, where one child node encapsulates the silver base of the light bulb, and another child node encapsulates the glass part of the light bulb.
- The parent node may also just serve as a means of grouping the children nodes. For example, four children might encapsulate data that renders four tires on a car.
- The children nodes can all be views of the same geometry at different levels of resolution.

  The parent node, in this example, switches the source of the display data between its child nodes according to, for example, how far the geometry is from the viewer. The bottom of the group contains geometry that is stored in several childless leaf nodes, such as pfGeode.

The entire scene graph represents all of the data in the database.

Scene Graph Traversers

Nodes in a scene graph can respond to some kind of traversal. A traverser runs over part or all of the hierarchy of nodes in the scene graph, which triggers some response. Not all nodes, however, respond to all traversals.

Examples of traversers include the CULL traverser, pfCull, and the DRAW traverser, pfDraw. pfNodes can use the OpenGL Performer default behavior, or define their own using callback mechanisms. For more information about callbacks, see "Customizing OpenGL Performer Traversals" on page 106.

The hierarchy of the scene graph specifies the order in which the nodes are processed by a traversal. The order is top down. For example, the numbers on the nodes in Figure 4-2 show one example of the order in which the traversal is applied to each node.

A traversal going from one node to another is said to traverse the scene graph. Figure 4-2 shows that the traversal was applied at the root node, pfScene, of the scene graph. A traversal applied at the root is passed throughout the entire scene graph. You can, however, apply the traversal to a subsection of the scene graph by applying it to any node beside the root node.

For more information about scene graphs, see Chapter 6, "Creating Scene Graphs."

Channels

A channel is equivalent to a camera moving throughout the scene. Whereas the scene graph encapsulates all of the visual data in the scene, the channel contains only that visual information that is visible to the viewer; the channel shows a slice of the scene from a specified perspective. The view culled by the channel is defined by:

- Camera position and orientation
- Viewing frustum

The channel provides a particular view of a scene, as shown in Figure 4-3.

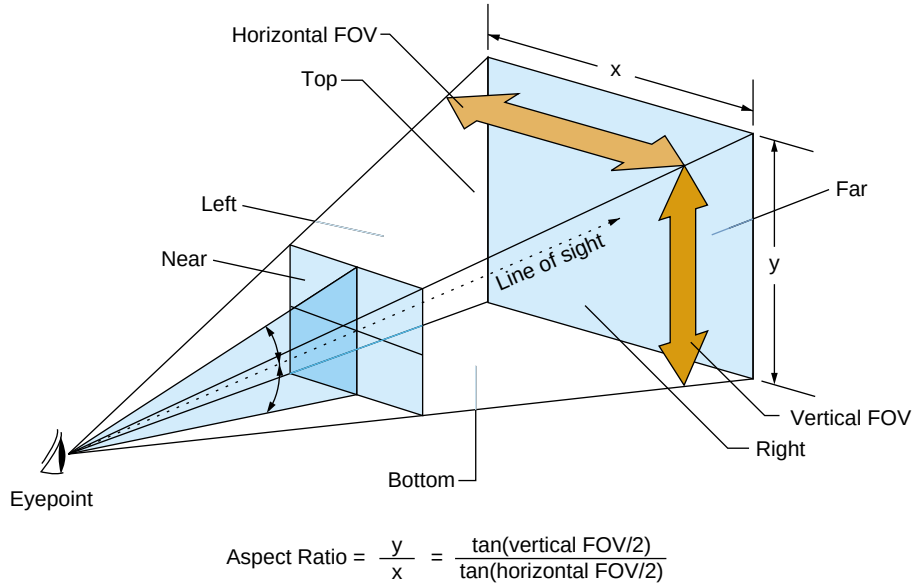

Figure 4-3 Camera with Viewing Volume

Note: OpenGL Performer allows you to create asymmetric frustums.

The *viewing volume* is the pyramid shown in Figure 4-3. The *frustum* is the truncated pyramid defined by:

- Near and far clipping planes.
- Horizontal and vertical fields of view.

The only geometries in view are those in the viewing frustum. Geometries in the scene graph are invisible when they are:

- Beyond the far clipping plane.
- Between the viewer and the near clipping plane.
- Outside the horizontal and vertical fields of view.

Each channel is associated with a scene graph; however, one scene graph may be associated with more than one channel.

For more information about channels, see Chapter 5, “Creating a Display with pfChannel.”

Pipe and Window

The pipe renders the visual data in the viewing frustum to a window. The pipe is the software abstraction of the hardware graphics pipeline.

Rendering the scene occurs in three stages:

1. APP—updates the location and look of geometries and updates the viewing location and orientation.
2. CULL—determines which geometries in the scene are visible (in the viewing frustum), taking occlusion into account.
3. DRAW—renders all visible geometries.

Each stage is potentially a separate process. For maximum performance, run each of these processes on a different CPU. When using three CPUs, OpenGL Performer can process three frames at the same time, as shown in Figure 4-4.

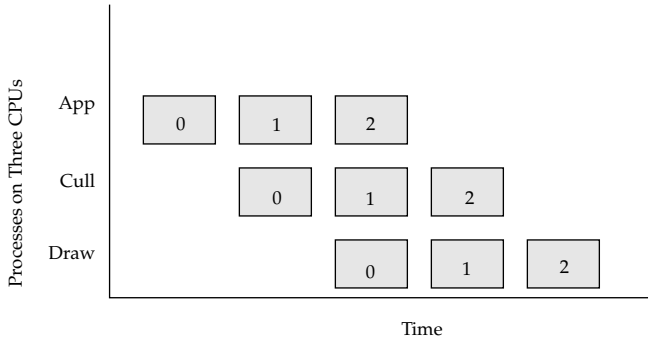

Figure 4-4 Multiprocessing Frames in the Pipe

Figure 4-4 shows how:

- Three frames are processed sequentially across three processes.
- The three processes running on three CPUs can process up to three frames of data concurrently, while the APP stage processes the third frame, the CULL stage processes the second frame, and the DRAW stage processes the second frame.

Most of your work is done in the APP stage, which updates the location of the geometries and the camera in the scene.

It is possible to customize the CULL and DRAW stages using callback functions. It is more common, however, to let OpenGL Performer handle those stages. For more information about customizing stages using callback functions, see "Customizing OpenGL Performer Traversals" on page 106.

Starting the Stages

Each stage runs as a separate traversal over the scene graph. Table 4-1 shows the classes that start each stage.

Table 4-1 Traversals Launched

Traversal	Launched By
APP	pfApp (from pfSync)
CULL	pfCull (from pfFrame)
DRAW	pfDraw (from pfFrame)
ISECT	pfNodeIsectSegs/pfChanNodeIsectSegs

The ISECT traversal searches for intersections. For more information, see Chapter 13, "Intersection Testing."

Rendering the Scene

When you call pfFrame, the CULL traversal generates a `libpr` display list of geometry and state commands, which describes the scene that is visible from a pfChannel.

- The DRAW traversal traverses the display list and sends commands to the Geometry Pipeline to generate the scene.

The `libpr` display list keeps pointers to user data. The list allows users to have dynamic data.

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.

Display Lists

`libpr` supports display lists, which contain and later execute `libpr` graphics commands. **pfNewDList()** creates and returns a handle to a new pfDispList. You can select a pfDispList 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. In display list mode, changes to the scene do not take effect until the next pfFrame is called.

It is not legal to have multiple pfDispLists open at the same 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. pfFrame, however, executes the display list automatically.

For more information on pfDispList, see the *OpenGL Performer Programmer's Guide*.

Parts of a Performer Application

The basic parts of your OpenGL Performer program include:

1. Initializing OpenGL Performer.
2. Acquiring a pipe.
3. Creating a channel and window.
4. Associating the channel with the appropriate window.
5. Loading the scene graph and associate it with the channel.
6. Positioning the channel(s) and updating the scene.
7. Calling pfFrame to draw the scene.
8. Creating the simulation loop to return to step 6.

Initializing Performer

To initialize OpenGL Performer, use the following call:

```
void pfInit(void);
```

Initializing OpenGL Performer causes the following:

- Sets up the shared memory arena for the three processes.
- Initializes the OpenGL Performer graphics state.

pfInit() must be the first method called in an OpenGL Performer application. pfConfig creates the additional, desired OpenGL Performer processes. You clean up by calling **pfExit()**, which exits the application and kills all OpenGL Performer processes.

Shared Memory Arena

Because all three processes can work on the same frame of visual data, a shared memory arena is required. OpenGL Performer uses shared memory arenas that can be accessed by separate processes. All OpenGL Performer processes need to access the scene graph, so all scene graph data must be in the shared memory arena. OpenGL Performer creates the arena for you in pfInit in a `libpf` application.

pfNodes and other `libpf` objects are automatically created in the shared memory arena. You can get the shared arena with pfGetSharedArena.

For more information about shared memory, see Chapter 19, “Performance Tuning and Debugging”, in the *OpenGL Performer Programmer’s Guide*.

Creating the Pipe, Channel, and Pipe Window

To create the pipe, channel, and pipe window, use the following calls:

```
// pfConfig must go first
pfConfig();

// Acquire handled pipe number 0.
pfPipe *pipe = pfGetPipe(0);

// Create the pipe window and associate it with the pipe.
pfPipeWindow *pwin0 = pfNewPWin(pipe);
```

```
// Create the channel and associate it with the pipe.
pfChannel *chan0 = pfNewChan(pipe);

// Associate the channel and pipe window so the channel is drawn in it.
pfAddChan(pwin0, chan0);

// Calls that cause the window to be opened at next pfFrame()
pfOpenPWin(pwin0);
pfFrame();
```

These methods configure the graphics pipeline, and fork all requested processes, including APP, CULL, and DRAW.

Loading the Scene Graph

To specify the path to the scene graph file, use the following method:

```
void pfFilePath(const char *pathName);
```

pathName is the complete path to the scene graph file. For more information about loading a scene graph, see "Loading a Scene Graph" on page 100.

Positioning the Channel

Use the following methods to position the channel:

```
void pfChanFOV(pfChannel* chan, float horiz, float vert);
void pfChanNearFar(pfChannel* chan, float near, float far);
```

The simulation loop, up to this point, has handled the APP process. To start the CULL and DRAW processes, use the following call:

```
void pfFrame(void);
```

Besides starting the CULL and DRAW processes, this call, along with pfSync, handles frame synchronization.

Creating the Simulation Loop

The simulation loop drives the application. Its actions are carried out in the APP process. The simulation loop repeats endlessly until the application exits.

Figure 4-5 shows that the simulation loop generally has three steps:

1. Update the appearance, shape, and location of the objects in the scene.
2. Update the position and orientation of the camera in the scene.
3. Redraw the scene.

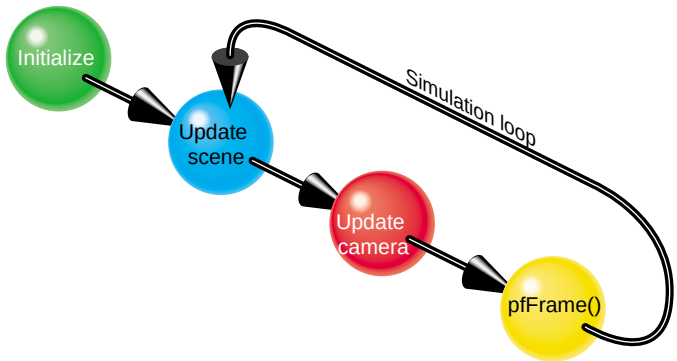

Figure 4-5 Simulation Loop

Example 4-1 shows pseudo code for a synchronization loop.

Example 4-1 Synchronization Loop

```
while(!finished)
{
    handleinput();
    updateScene();
    anyCriticalUpdate();
    pfFrame();
}
```

Inputting and Reading User Events

Performer applications are interactive and use the OpenGL/X Window System on IRIX and Linux systems.

User input devices are unlimited, but OpenGL Performer provides utilities for handling the following:

- Keyboard
- Mouse
- Track ball
- Flybox

Other input can come from the following:

- Network
- Reflective memory

Implementing User Input with Window Events

To implement user input, you need to:

1. Initialize the utility library.
2. Set the window type.
3. Enable user input on the window.
4. Collect the window events in a forked process.

Initializing the Utility Library

The utility library, `libpfutil`, provides input handling utilities. Initialize it by using the following call:

```
void pfuInitUtil(void);
```

Clean it up by using the following call:

```
void pfuExitUtil();
```

Enabling User Input

To enable user input, you can use the following method:

```
void pfuInitInput( *pwin, int mode);
```

where *pwin* points to the pfPipeWindow where the user enters information, and *mode* is one of two tokens:

- PFUINPUT_X
- PFUINPUT_X_NOFORK

You can set the window with the following method:

```
void pfPWinType( pwin, type);
```

where *pwin* points to the pfPipeWindow where the user enters information and *type* is one of the following tokens:

- PFPWIN_TYPE_X
- PFPWIN_TYPE_X_NOFORK

Note: On Windows, the user events are not processed in a separate process. Hence, use PFUINPUT_X_NOFORK and PFPWIN_TYPE_X_NOFORK.

To clean up when user input is finished, use the following method:

```
pfuExitInput();
```

Retrieving User Events

OpenGL Performer provides utilities for asynchronously calculating window system events and returning them to the application. User events are stored in a pfuEventStream object, which is a queue. To retrieve keyboard and mouse events, use the following methods, respectively:

```
pfuGetEvents (event);
pfuGetMouse(mouse);
```

where *event* and *mouse* are pointers to keyboard events and a pfuMouse structure, respectively.

To complete the implementation, you must respond to the following events:

- Examine the keyboard input and take appropriate actions.
- Use pfiXformer to handle the mouse events.

OpenGL Performer provides a routine for examining keyboard input. You only need to add code that takes appropriate actions in response to the input, as shown in Example 4-2.

Example 4-2 Handling Keyboard Input

```
void handleEvents(void)
(

    extern pfuEventStream events;
    pfuEventStream *pEvents = &events;

    // get events and number of events pfuGetEvents(&events);
    numDevs = pEvents->numDevs;

    // process each of the events; dev is the kind of event, val is
    // its value, such as a keyboard event with an ASCii value of 27.
    for ( j=0; j = numDevs; ++j) (
        dev = pEvents->devQ[j];
        val = pEvents->devVal[j];

        if ( pEvents->devCount[dev] > 0) {
            switch ( dev ) {
            ...

            // process keyboard input
            case PFUDEV_KEYBD:
                for ( i=0; i < pEvents->numKeys; ++i ) {
                key = pEvents->keyQ[i];
                if ( pEvents->keyCount[ key ] ) {

                    switch [key ] {

                    case 27:
                    // escape key; exit program
                    exitFlag = 1; break;

                    case ‘h’:
                    // print help
                    printHelp(progName);
                    break; ...
```

Creating a Display with pfChannel

A pfChannel is a view into a scene graph based on the following:

- Location and orientation of the camera in the scene
- Viewing frustum

A pfPipe can display one or more channels in one or more windows, as shown in Figure 5-1. Each pfChannel in a window corresponds to a *viewport*.

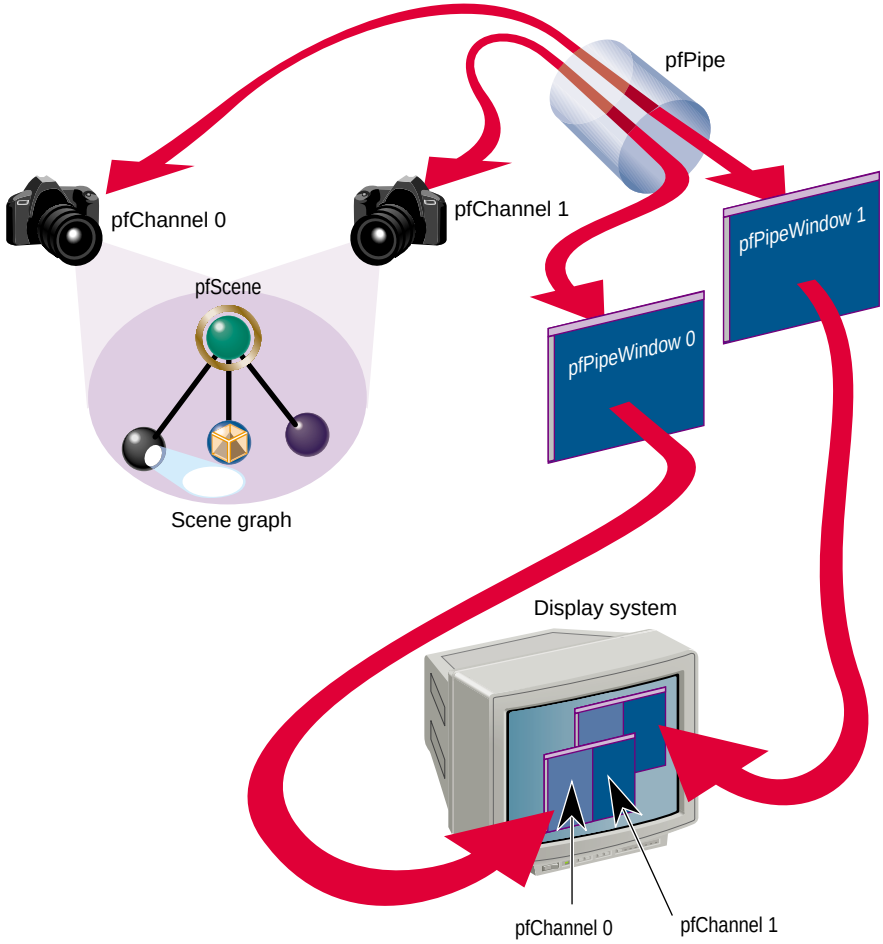

Figure 5-1 Multiple Windows, Multiple Channels

This chapter discusses, in the following sections, many of the important classes that constitute the process of taking data from a scene graph database and rendering it on a display system:

- "Creating and Configuring a pfChannel" on page 77
- "Initializing the pfChannel View" on page 79
- "Channel Callbacks" on page 85

Creating and Configuring a pfChannel

To use a pfChannel:

1. "Acquiring a pfPipe" on page 77.
2. "Creating and Configuring a pfPipeWindow" on page 78. (Optional)
3. "Creating a pfChannel Rendered by a pfPipe" on page 78.
4. "Attaching a pfScene to the pfChannel" on page 78.
5. "Configuring a Viewport for the pfChannel" on page 78. (Optional)
6. "Defining the Viewing Frustum" on page 81.

Note: Steps 4, 5, and 6 can be completed in any order.

The following sections explain this procedure.

Acquiring a pfPipe

A pfPipe is a graphics pipeline that renders one or more pfChannels into one or more pfPipeWindows, as shown in Figure 5-1.

The pfPipe is the software abstraction of the hardware graphics pipeline. You can create as many pfPipe objects as you like. For optimal performance, use only one pfPipe object per hardware graphics pipeline.

To acquire a handle to a pfPipe object, use the following pfPipe method:

```
pfPipe *pipe = pfGetPipe(0);
```

Creating a pfChannel Rendered by a pfPipe

To create a pfChannel, use the following constructor:

```
pfChannel *channel = new pfNewChan(pipe);
```

where *pipe* is a pointer to a pfPipe.

The pfChannel is automatically associated with the first pfPipeWindow in the pfPipe. If a pfPipeWindow is not explicitly created, one is generated automatically and set to be fully screened.

Creating and Configuring a pfPipeWindow

To create and configure a pfPipeWindow in which the pfPipe displays its rendering, use the following method:

```
pfPipeWindow* pfNewPWin(pfPipe *pipe);
```

where *pipe* is a pointer to a pfPipe.

Use the pfPipeWindow methods to configure a pfPipeWindow.

Attaching a pfScene to the pfChannel

To attach a scene to the pfChannel, use the following method:

```
void pfChanScene(pfChannel *chan, pfScene *scene);
```

where *chan* and *scene* are the pfChannel to a scene to connect.

Configuring a Viewport for the pfChannel

The viewport is that portion of the pfWindow in which the pfChannel is displayed, as shown in Figure 5-1. If you do not configure the viewport, the pfChannel defaults to displaying in the entire pfWindow.

You can create and modify the viewport of a pfChannel using the following line of code:

```
void pfChanViewport(pfChannel *chan, float left, float right,
```

```
    float bottom, float top);
```

chan is the pfChannel associated with the viewport.

left and *right* specify the X coordinates from the left side to the right side of the viewport. Values are clamped between 0.0 and 1.0, where 1.0 is the entire width of the pfWindow.

bottom and *top* specify the Y coordinates from the bottom to the top of the viewport. Values are clamped between 0.0 and 1.0, where 1.0 is the entire height of the pfWindow.

Creating a Background for a pfChannel

pfEarthSky objects draw sky, horizon, and ground in different weather conditions.

To display the background, follow these steps:

1. Use pfClear to clear the buffers in the current graphics window.
2. Use pfNewESky to create a new pfEarthSky object.
3. Use **pfChannel::pfChanESky()** to attach the pfEarthSky to a pfChannel.

pfEarthSky is called automatically in the DRAW process, unless a DRAW callback is present, in which case it must be explicitly called using pfClearChan.

Initializing the pfChannel View

You might like to start the size of the view frustrum so that the shape in it is completely in view. To determine the size of a shape, you retrieve the bounding sphere for the shape. The bounding sphere is a sphere that encloses a shape and approximates its size, as shown in Figure 5-2.

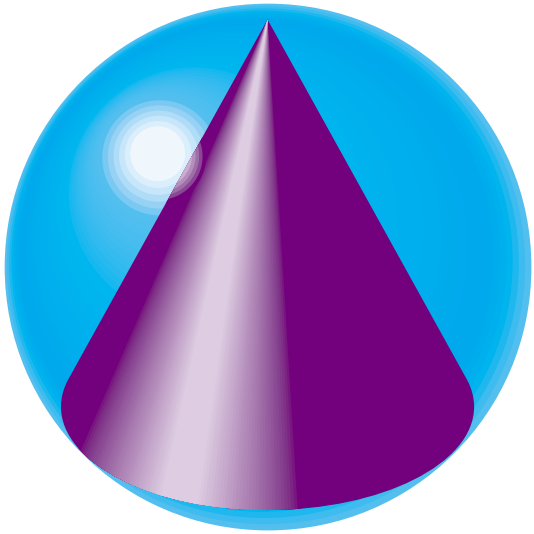

Figure 5-2 Bounding Sphere

To return the size of a bounding sphere, use the following method:

```
int pfGetNodeBSphere( pfNode *node, pfSphere *sphere);
```

This method returns the bounding sphere, *sphere*, for the node, *node*. The bounding sphere is returned as a pfSphere, defined as:

```
typedef struct {
    pfVec3 center;
    float radius;
} pfSphere;
```

Note: The bounding spheres are often slightly larger than the size of the geometry.

Use the dimensions of the bounding sphere as a guideline for the starting size of the view frustum, for example:

```
pfGetNodeBSphere(shapeNode, &Bsphere);
distanceToShape = 2.0f * Bsphere.radius;
pfChanNearFar(chan, 1.0f, 10.0f * Bsphere.radius);
```

```
float sceneSize = 2*bsphere.radius;

    /* Set initial view to be “in front” of scene */
    /* first put view point at center of sphere */
    PFCOPY_VEC3(Shared->view.xyz, bsphere.center);

    /* then back up so all is visible */
    Shared->view.xyz[PF_Y] -=      sceneSize;
    Shared->view.xyz[PF_Z] += 0.25f*sceneSize;

    /* look up the Y axis */
    pfSetVec3(Shared->view.hpr, 0.0f, 0.0f, 0.0f);

    pfChanView(chan, Shared->view.xyz, Shared->view.hpr);
```

Bounding Volumes

The following geometries are used as bounding volumes for the following classes:

- pfSpheres are used as bounding volumes for pfNodes.
- pfBoxes are used as bounding volumes for pfGeoSets.
- pfCylinders are used as bounding volumes for intersection rays.

Defining the Viewing Frustum

The viewing frustum is defined by the following:

- “Near and Far Clip Planes” on page 82
- “Height and Width of the View Frustum” on page 83
- “Direction and Position of the View” on page 83

These parameters are shown in Figure 5-3.

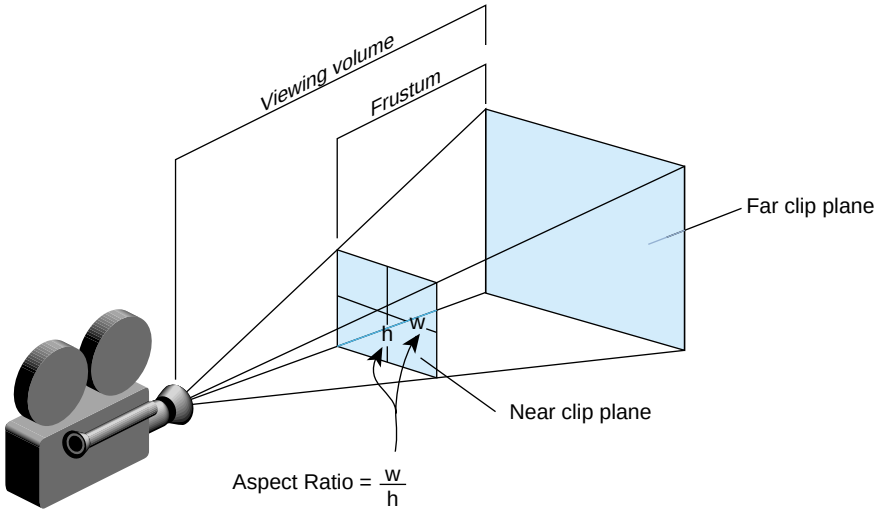

Figure 5-3 Viewing Frustum

The following sections explain how to set these parameters.

Near and Far Clip Planes

To set up the viewing frustum, use the following method:

```
void pfChanNearFar( pfChannel *chan, float near, float far);
```

chan is the pfChannel, *near* and *far* define the distances to the near and far clipping planes of the viewing frustum.

Shapes closer to the camera than the near clip plane or further than the far clip plane are not rendered. The default values for each clip plane are:

- Near clip plane = 1.0
- Far clip plane = 1000.0

The near clip plane must lie between 0.0 and the far clip plane.

Tip: Moving the near clip plane close to the origin degrades Z-buffer precision.

Height and Width of the View Frustum

To set up the height and width of viewing frustum, use the following method:

```
void pfChanFOV( pfChannel *chan, float horiz, float vert);
```

where *chan* is the pfChannel and *horiz* and *vert*, expressed in degrees, define the horizontal and vertical dimensions, respectively, of the view frustum, as shown in Figure 5-3. The default values are:

- *horiz* = 45.0
- *vert* = 0.0

If one angle is less than or equal to 0.0 degrees, that dimension is computed using the other angle and the viewport aspect ratio. Generally, you should only specify one of the angles so that the other is determined automatically to fit into viewport without distortion.

Direction and Position of the View

To set up the direction and position of the viewing frustum, use the following method:

```
void pfChanView( pfChannel *chan, pfVec3 xyz, pfVec3 hpr);
```

where

- *chan* is the pfChannel.
- *xyz* is the position of the camera in world space coordinates.
- *hpr* is the heading, pitch, and roll of the camera, specified in degrees.

The heading, pitch, and roll values account for the rotation of the camera in any of the three dimensions, as shown in Figure 5-4.

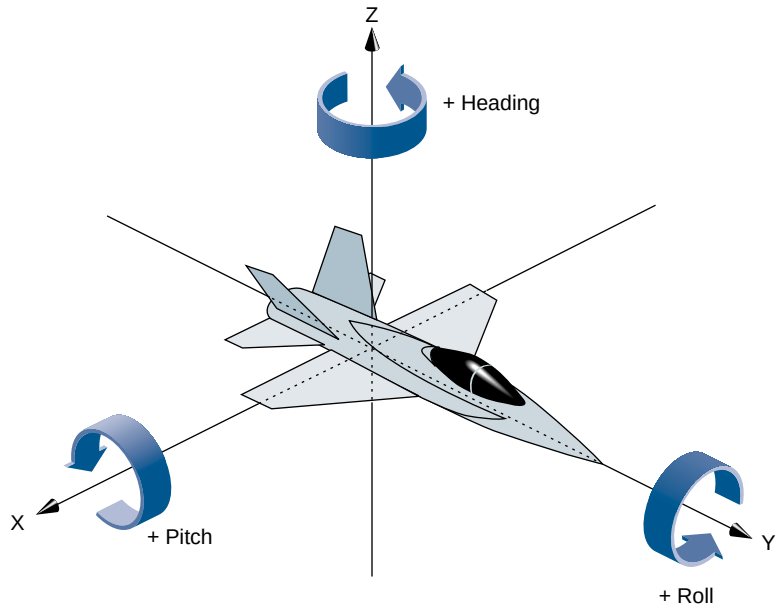

Figure 5-4 Heading, Pitch, and Roll Values

The starting *hpr* is oriented at the origin looking down the Y-axis.

OpenGL Performer also includes a pfCoord data type that defines the location and the rotation values.

```
typedef float pfVec3[3];

typedef struct (
    pfVec3 xyz;
    pfVec3 hpr;
}pfCoord;
```

Note: The multiplication order is roll, pitch, and then heading.

Channel Callbacks

CULL and DRAW traversals are executed for each channel on a pfPipe whenever you call pfFrame. If you want to modify the default behavior when pfFrame is called on a channel, you can use callback functions.

For example, if you have special knowledge that the default CULL process cannot know, customize the behavior of the channel when CULL traverses the channel. One example is if you know you are inside a house, you would cull everything outside the house. The default CULL process would not necessarily produce this result.

To set up a channel callback function, use the following method:

```
void pfChanTravFunc(pfChannel* chan, int trav, pfChanFuncType func);
```

chan is the channel for which you are setting up the callback function.

trav is the kind of traversal that triggers the callback function. Possible values include:

- PFTRAV_CULL
- PFTRAV_DRAW

func is the callback function called when the specified traversal, *trav*, evaluates the channel, *chan*.

The default behavior is to call:

- pfCull from the CULL callback
- pfClearChan and pfDraw from the DRAW callback

You can either ignore or add to this behavior.

Using Passthrough Data

The data derived from a channel traversal callback function must be sent down the graphics pipeline at frame boundaries. To synchronize the use of channel callback data, follow these steps:

1. Allocate memory and associate it with a channel.
2. Mark the data in the allocated memory as ready to be passed down the graphics pipeline at the next pfFrame call.

The following methods accomplish those tasks:

```
void * pfAllocChanData(pfChannel* chan, int size);
void pfPassChanData(pfChannel* chan);
```

size is the number of bytes of memory to allocate and associate with the channel, *chan*.

Changes are passed down the graphics pipeline only. For example, changes in the CULL process are not seen in the APP process.

Channel Callback Example

Example 5-1 shows how to implement channel callbacks using pass through data. The example assumes that the CULL and DRAW callbacks have already been set up.

Example 5-1 Channel Callback Using Passthrough Data

```
typedef struct {
    int frameCount;
} PassData;

PassData *data = (PassData *)pfAllocChanData(chan, sizeof(PassData));
...
while (...) {
    data->frameCount = pfSync();
    pfPassChan Data(chan);
    pfFrame();
}

void cullChan(pfShannel *chan, void *data)
{
    // Changes made to data will be seen in channel DRAW callback.

    PassData *pass = (PassData *)data;
    pass->frameCount++;
    ...
}
```

Using Multiple Channels

Multiple channels can be connected to a single pfPipe; a single pfChannel, however, cannot be connected to more than one pfPipe. pfPipe maintains a list of channels attached to it.

Each pfChannel is rendered in its own viewport, as shown in Figure 5-5.

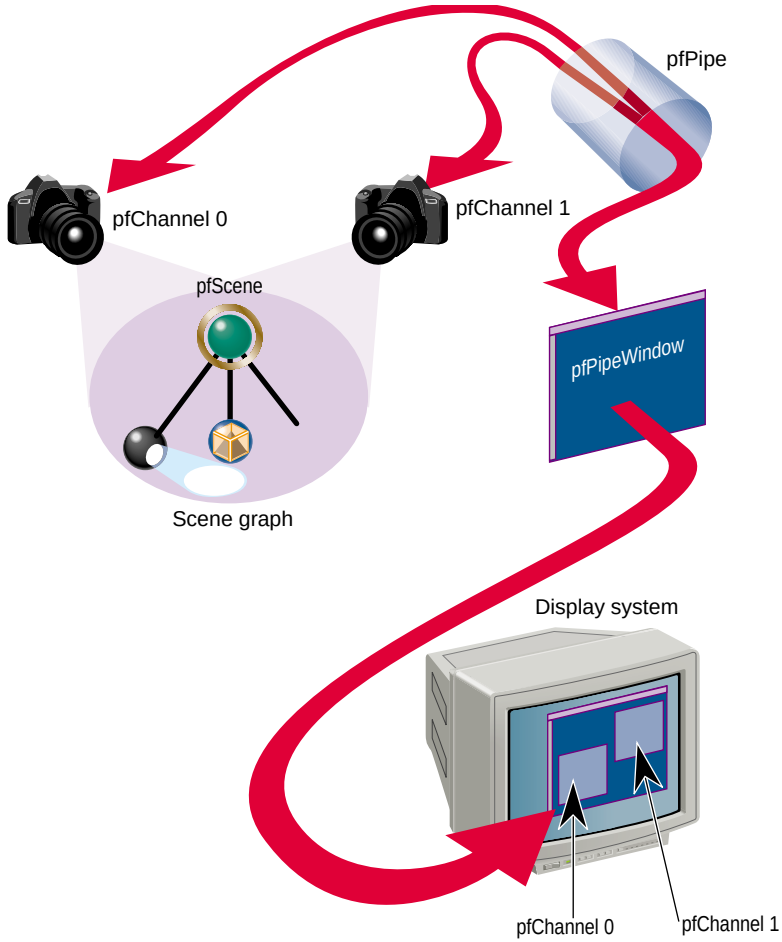

Figure 5-5 Multiple Channels

In Figure 5-5, pfChannel 0 is rendered in viewport 0, and pfChannel 1 is rendered in viewport 1. The channels are rendered in the order they are created.

For information about rendering channels in their own viewport, see "Configuring a Viewport for the pfChannel" on page 78.

Grouping Channels

At times you may want to have different views of the same part of the scene. For example, each pfChannel might show the same scene at different LOD levels. These two views of the same scene require that the channels showing each view have the same attributes.

OpenGL Performer makes it easy for different channels to share attributes by creating pfChannel groups. One pfChannel is chosen as the master pfChannel and the other channels in the group are attached to the master, as follows:

```
int pfAttachChan(pfChannel *master, pfChannel *chan);
```

master is the pfChannel whose attributes are used for all channels attached to it. *chan* is a different pfChannel that is dependent upon the *master* pfChannel; *chan* uses the *master* pfChannel's attributes.

You can add as many channels to the group as you like by repeating the **pfAttachChan()** method.

Choosing the Attributes to Share

By default, the channels in a group use the master pfChannel's attributes except for:

- Viewport
- View offsets, explained in "Using View Offsets" on page 90
- Hardware swap buffers signal (for multichassis sync)

You can, however, specify other attributes that you do not want a pfChannel to derive from the master by setting the bits in a mask. By default, all of the bits in the mask are ON. To unshare attributes, follow these steps:

1. Get the pfChannel's mask, using:

   ```
   uint pfGetChanShare(pfChannel *chan);
   ```

2. Unset the bit for the attribute you do not want to be shared with the master pfChannel.

3. Set the mask, using:

   ```
   void pfChanShare(pfChannel *chan, uint mask);
   ```

The following code segment shows an implementation of this procedure in which the attribute, near and far planes, is unset.

```
mask = pfGetChanShare(chan);
mask &= !PFCHAN_NEARFAR;
pfChanShare(chan, mask);
```

In this example, the view frustum of the pfChannel may be different from that of the master pfChannel.

Attribute Mask

Table 5-1 lists the pfChannel attributes that can be shared.

Table 5-1 pfChannel Attributes

pfChannel property	Description
PFCHAN_FOV	Field of view angles
PFCHAN_NEARFAR	Near and far clip planes
PFCHAN_VIEW	View position
PFCHAN_VIEW_OFFSETS	*xyz, hpr* offsets from master viewpoint
PFCHAN_VIEWPORT	Viewport
PFCHAN_SCENE	Scene
PFCHAN_EARTHSKY	Earth-sky model
PFCHAN_STRESS	Stress filter parameters
PFCHAN_LOD	Level of detail modifiers
PFCHAN_SWAPBUFFERS	Signal to swap

Table 5-1 pfChannel Attributes **(continued)**

pfChannel property	Description
PFCHAN_SWAPBUFFERS_HW	Signal to swap (multipipe)
PFCHAN_STATS_DRAWMODE	Statistics graph characteristics
PFCHAN_APPFUNC	Application callback
PFCHAN_CULLFUNC	CULL callback
PFCHAN_DRAWFUNC	DRAW callback

Using View Offsets

Although a pfChannel might look at the same scene as that seen by the master pfChannel, you might like to orient the pfChannel differently. For example, you might like the master pfChannel to show one view of a scene and a slave pfChannel to show a view of adjacent scenery so that when the two channels are projected side by side, you have a wide view of the scene.

The view offset can be set for each pfChannel. If it is not set, the offset is zero, which means the master and slave channels are in the same location and orientation.

To specify how a slave's pfChannel view is different from the master pfChannel View, use the following method:

```
void pfChanViewOffsets(pfChannel* chan, pfVec3 xyz, pfVec3 hpr);
```

chan specifies the pfChannel to offset.

xyz specifies the 3D coordinates, relative to the master pfChannel, where the slave pfChannel is located.

hpr specifies the rotation of the pfChannel, relative to the master pfChannel, where h, p, and r are the degrees of rotation about the x, y, and z axes, respectively.

For example, if you want the slave pfChannel to be 100 units above the master pfChannel pointed down, the values for x, y, and z would be (0, 0, 100) and the values for h, p, and r would be (0, -90, 0).

The Z-axis is oriented vertically to the ground, as shown in Figure 5-6.

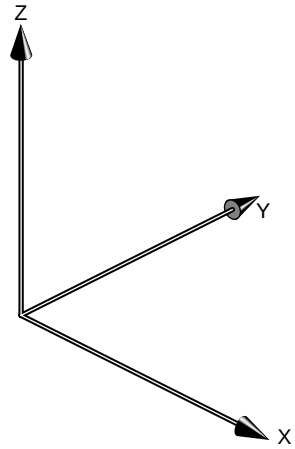

Figure 5-6 Axes Orientation in Performer

The starting orientation is located at the origin and pointed down the Y-axis.

Multiple Pipes

You may find it appropriate to display your data over more than one display system. For example, you might want to present the left side and right side of a scene on two different monitors. The CULL and DRAW stages are specific to each pfPipe object; the APP stage, however, is shared by both pfPipe objects, as shown in Figure 5-7.

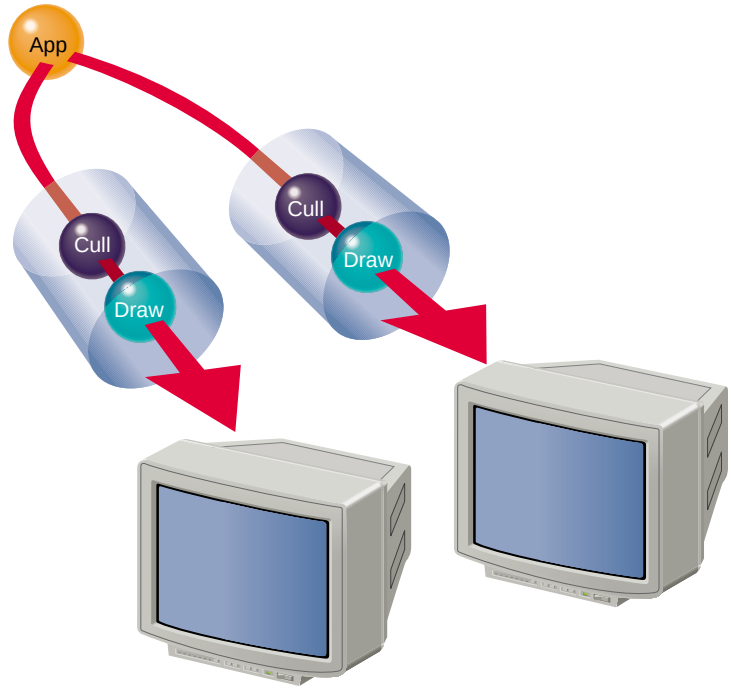

Figure 5-7 Pipe Stages

Setting the Multiprocessing Configuration

pfMultiprocess controls the multiprocessing configuration of a pfPipe. The CULL and DRAW stages can run as a single process or, for improved performance, run as separate processes, according to the value passed to pfMultiprocess. For example, to run the APP, CULL and DRAW stages as separate processes, use the following line of code:

```
pfMultiprocess(PFMP_APP_CULL_DRAW);
```

You must call pfMultiprocess between pfInit and pfConfig. If you do not, OpenGL Performer creates a multiprocessing configuration automatically based on the number of CPUs in the run-time hardware.

For more information about multiprocessing, see Chapter 11, “Multiprocessing.”

Creating Multiple pfPipes

To create multiple pfPipe objects, use the following pfPipe method:

```
void pfMultipipe(int npipes);
```

npipes is the number of pfPipe objects to create.

Call pfMultipipe between pfInit and pfConfig. By default, there is just one pfPipe.

For more information about multiprocessing, see Chapter 11, “Multiprocessing.”

Creating Scene Graphs

A scene graph holds the data that defines a virtual world. The scene graph includes low-level descriptions of object geometry and their appearance, as well as higher-level, spatial information, such as specifying positions, animations, and transformations of objects, as well as additional application-specific data.

Scene graph data is encapsulated in many different types of nodes. One node might contain the geometric data of an object; another node might contain the transformation to orient and position it in the virtual world.

Nodes are associated in a hierarchy that is a directed, acyclic graph. OpenGL Performer and your application can act on the scene graph to perform various complex operations efficiently, such as database intersection and rendering scenes.

This chapter describes how to create, change, load, and save scene graphs, in the following sections:

What Is a Node?

A node is a scene graph building block; a pfNode is the abstract class from which OpenGL Performer nodes are inherited. A complete OpenGL Performer scene graph is one that is rooted by a pfScene node. The most common type of node is the pfGroup

node, which can take an arbitrary number of child nodes. Other node types are more discriminating, and provide structure and semantics to the operations that process them.

pfNodes are opaque classes; methods are used to get and set all member fields. For example, a few of the methods setting some node fields are included in Table 6-1.

Table 6-1 Examples of Node Fields

Node	Field	Type	Method	Description
pfSwitch	Val	float	pfSwitchVal	Selects the active child under a pfSwitch node.
pfLOD	Center	pfVec3	pfLODCenter	Sets the center for LOD evaluation.
pfLayer	Mode	int	pfLayerMode	Selects a method of coplanar object layering.

Because it is subclassed from pfUpdatable, pfNode is automatically multibuffered for multiprocessing. This feature enables pfNode subclasses to be edited safely from the application processes while OpenGL Performer is using them for scene graph operations in other processes.

Nodes in OpenGL Performer are used for describing a virtual world. Objects associated with viewing the world, such as pfChannels, are not nodes and are not placed in the scene graph. Only pfNode and its subclasses can be placed directly in the scene graph and only some of those subclasses can take child nodes.

Node Attributes

All nodes have the following attributes:

- Parent list — node(s) from which the node is subclassed.
- Bounding volume—volume, sphere, box, or cylinder that completely surrounds a shape and is roughly equivalent to the size of the shape. A bounding volume makes such things as collisions and culling faster to compute.
- Name— name of the node.
- Type— node type.
- Traversal masks—directs a traversal to a subgraph of nodes.
- Callback functions and data—callback functions enable the programmer to customize the behavior of certain traversals for specific nodes.

Scene Graph Nodes

The two most general classifications of node functionality are:

- Group nodes—associate nodes into hierarchies.
- Leaf nodes—contain all the descriptive data of objects in the virtual world used to render them.

Group Nodes

Only group nodes can have child nodes. Each child node has an index number; the first child added to a group node has an index number of 0, the next child added has an index number of 1, and so on. The group node keeps a list of its child nodes.

A child node may have multiple parent nodes. For example, a node encapsulating a wheel might have four parent nodes, each translating the shape of the wheel to a different place in the scene (on a car), as shown in Figure 6-1.

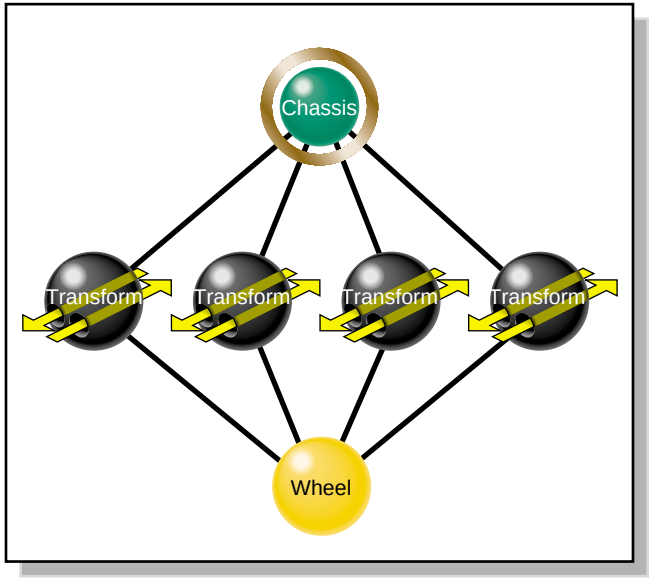

Figure 6-1 Multiple Parent Nodes

OpenGL Performer Group Nodes

In OpenGL Performer, the pfNodes inherited from pfGroup are:

- pfScene—root node of a scene graph.
- pfLOD—its children represent the same shape but at different levels of resolution (LOD).
- pfSCS—stores a static coordinate system transformation in which to place its children.
- pfDCS—contains a changeable coordinate system in which to place its children.
- pfFCS—contains a pfFlux for holding a coordinate system transformation that is computed by an asynchronous process.
- pfSwitch—node that directs a traversal to one, all, or none of the child nodes.
- pfSequence—node that directs a traversal to each of its child nodes one at a time, sequentially.
- pfLayer—groups coplanar polygons: the first child is the base and the following children layer on top of it and one another.
- pfPartition—group that optimizes very flat terrains.

Leaf Nodes

Leaf nodes contain the descriptive values used to render all the visual elements in the virtual world. Leaf nodes cannot have child nodes.

Special leaf nodes in OpenGL Performer include:

- pfGeode—encapsulates general geometry in the scene graph.
- pfLightSource—contains global light sources.
- pfASD—contains a continuous morphing LOD surface.
- pfBillboard—makes a slice of geometry turn to always face the viewer, which reduces the amount of rendering necessary to view a shape.
- pfText—incorporates 3D text into a scene graph.

Creating a Scene Graph

Creating a scene graph is an iterative process of adding child nodes (leaf and group) to group nodes. Eventually, you create a tree rooted at a pfScene node.

Creating and Attaching the pfScene Node

The root node, pfScene, is the node at the "top" of the scene graph hierarchy. pfScene is a group node because child nodes must be added to it. When a traversal is applied to it, the traversal is (potentially) passed to all other nodes in the scene graph.

You create a pfScene using the following method:

```
pfScene* root = pfNewScene();
```

pfScene nodes are attached to a pfChannel using the following method:

```
pfChanScene(chan, root);
```

A pfChannel provides a view of the geometric objects defined in the scene graph. For more information about pfChannel, see Chapter 5, "Creating a Display with pfChannel."

Adding Nodes in a Scene Graph

You can start anywhere in the hierarchy to create the scene graph. To create the hierarchy by follow these steps:

1. Create a group node and a child node using lines of code similar to the following:

   ```
   pfGroup* myGroup = pfNewGroup();
   pfASD* childNode = pfNewASD();
   ```

2. Add child nodes to the group node, as follows:

   ```
   pfAddChild(myGroup, childNode);
   ```

You continue making the hierarchy by repeating these steps.

Removing Nodes from a Scene Graph

To remove a node from a scene graph, use the following method:

```
pfRemoveChild(GroupNode, removeNode)
```

pfRemoveChild() returns 0 if the node is not a child of the specified group node.

When a child is removed, the index numbers of the remaining children are shifted so there is no discontinuity; all index numbers greater than the index number removed are decremented by one.

To find a specific node in a scene graph based on name and type, use **pfFindNode()**.

Arrangement of Nodes

The hierarchy of nodes is determined by the order in which you add nodes to one another. For example, if you start with the root node, called the pfScene node, and add a child node to it, it would appear in the scene graph directly below the pfScene node.

There are no rules for grouping nodes. However, there are some important guidelines that affect the performance of an application:

- Group nodes together for spatial coherence—put objects that are in the same basic location under the same group node.
- Rather than extend one object, such as a runway, across the entire scene, break static objects into multiple pieces in pfGeoSets or pfNodes so that parts of the object can be culled. In this way the culling traversal does not have to consider too many nodes at any one level in the scene graph.
- To reduce memory usage, for a potential minor performance cost, encapsulate in a separate node any objects that are used repeatedly. For example, a single node encapsulating a wheel can be referenced four times when creating a car, rather than using four nodes to encapsulate a wheel.
- In a vertical hierarchy, place all nodes comprising an object; if the top node of the shape is culled, the remaining nodes in the hierarchy of the shape are not evaluated.

Loading a Scene Graph

A scene graph you create might contain thousands of nodes; for that reason, they are retained on disk. The nodes are paged into memory according to the location of the

viewer, only those parts of the scene close to the viewer are in system memory. For more information on paging, see Chapter 12, “Database Paging.”

The explicit arrangement of data in the scene graph depends on the format used. Formats are identified by the extensions to the filenames, for example, Wavefront files use `.obj` and Workbench files use `.dwb`.

To load a scene graph file, pass the name of the file to **pfdLoadFile()**:

```
pfNode *pfdLoadFile(const char *filename);
```

filename is the name of the scene graph or subgraph database file.

pfdLoadFile() loads scene graph data at run time from the disk and constructs a graph from the data, as shown in Figure 6-2.

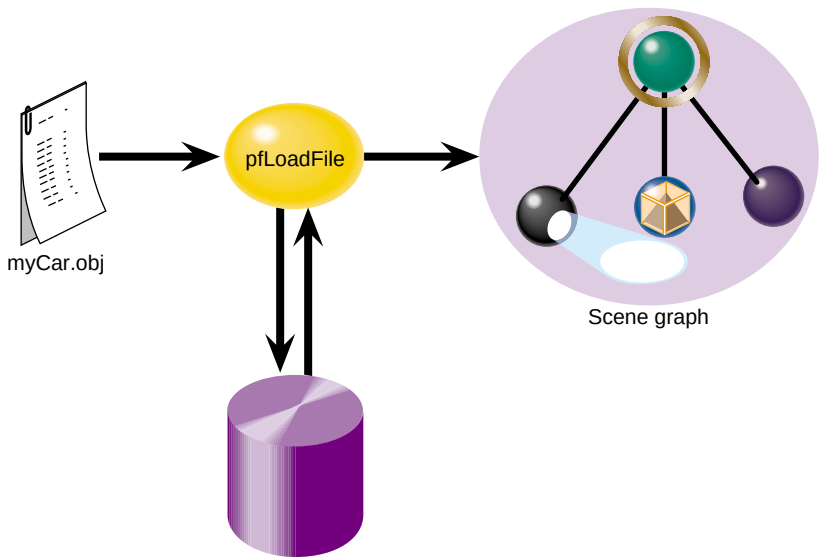

Figure 6-2 Loading Scene Graphs

pfdLoadFile() performs a run-time search for a DSO to load the file based on the filename suffix and calls the load routine **pfdLoadFile_*xxx*()** for the given database format. This mechanism allows OpenGL Performer to support an unlimited number of formats and to load new files in new formats or to load multiple formats at any time. The OpenGL Performer distribution includes a large number of file loaders.

Table 6-2 lists some of the more common file formats.

Table 6-2 Supported Scene Graph File Formats

Modeler	File Name Extension
Alias	Wavefront	`.obj`
3D Studio	`.3ds`
Coryphaeus	`.dwb`
Multigen	`.flt`
Inventor	`.iv`
Lightscape	`.lsa, .lsb`
Performer (native)	`.pfa, .pfb`

To see a complete list of supported formats, see the *OpenGL Performer Programmer's Guide*.

Finding Scene Graph Files

OpenGL Performer automatically looks for the file specified in **pfdLoadFile()** in the following directories in the following order:

1. Current directory.
2. Directories specified by the *PFPATH* environment variable.
3. Directories specified by **pfFilePath()** or **pfFilePathv()**.

   The function **pfFilePathv()** is the preferred function. See section "Setting the Search Path for Database Files" on page 235 for more details.

The last valid directory takes precedence over any before it. For example, if you had two versions of mySceneGraph.pif, one in the current directory and another in the directory specified by **pfdLoadFile()**, the version in **pfdLoadFile()** would be loaded.

Saving a Scene Graph

To save a scene graph or part of one, `libpfpfb` supports the following method:

```
int pfdStoreFile(pfNode *root, const char *filename)
```

root is the pfScene node or the top of the subgraph that you want to save.

filename is the name of the file in which the scene graph is stored. The same run-time search mechanism used for **pfdLoadFile()** is also used for **pfdStoreFile()** to find a file format writer for the requested format. To use **pfdStoreFile()** for run-time database paging and to use the fast OpenGL Performer paging format, the extension for the filename should be `.pfb`.

For more information about the OpenGL Performer binary format (`.pfb`), see "Optimizing File Loading" on page 223.

Scene Graph Traversals

A traversal is a method applied to (potentially) every node in a scene graph. Each node type responds in its own way by implementing a method call. For example, a common traversal culls the scene. Each pfNode implements a **cull()** method so the node can respond to the traversal. Individual node instances can further customize traversal behavior with their own callbacks.

Some nodes, called group nodes, simply pass the traversal to other nodes. In some cases (pfSwitch, pfLOD), the group node passes the traversal only to selected children nodes.

Other nodes, called leaf nodes, such as a pfGeode node, either encapsulate geometry to be rendered or represent significant computation, such as pfASD.

Pipelined Traversals

Several standard traversal operations are usually necessary for basic application operation and for the efficient rendering of a scene. OpenGL Performer provides automatic and transparent mechanisms for utilizing pipelined and parallel multiprocessing for handling these different traversals. The following processes can be

created by OpenGL Performer for the purpose of handling a specific traversal with its own effective copy of the scene graph nodes.

- APP—user traversal for updating the values in the nodes.
- CULL—evaluates application settings and eliminates the processing of any nodes out of view.
- DRAW—renders the culled scene graph.
- ISECT—intersects a set of line segments with the scene graph.
- DBASE—loads new database and deletes pieces no longer needed.

Each process is allotted its own memory space and acts on the scene graph nodes that reside in the shared memory arena, as shown in Figure 6-3. The scene graph geometry, and most of the actual scene graph data, is by default shared across the different processes and is not set up for multiprocessing. For multiprocessed geometry data, you should use the pfFlux object. See Chapter 14, "Dynamic Data," in the *OpenGL Performer Programmer's Guide*.

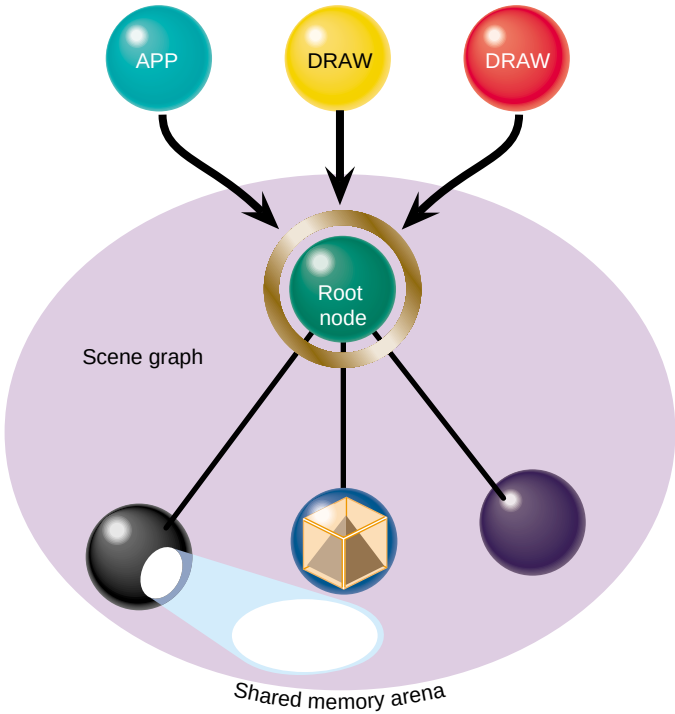

Figure 6-3 Processes Acting on Scene Graph

Traversal Order

Scene graphs are traversed in a depth-first, left-to-right order, as shown in Figure 6-4. At each node, some default behavior occurs. For example, a CULL stage starts a bounding sphere test to see whether the node is within the viewing frustum. Custom user pre and post-traversal callbacks on nodes are called as nodes are entered and exited.

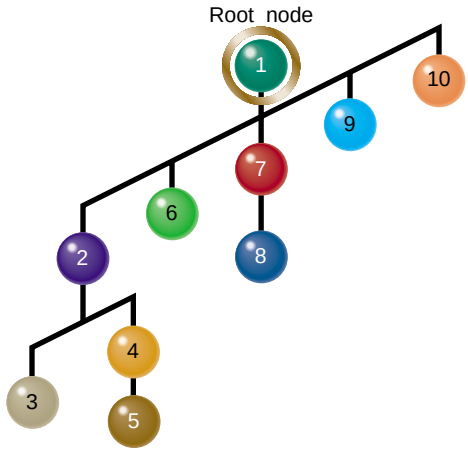

Figure 6-4 Scene Graph Traversal Flow

Customizing OpenGL Performer Traversals

In addition to providing callback functions for channels, discussed in “Channel Callbacks” on page 85, you can also customize the behavior of nodes by setting up callback functions for them.

You can customize the default behavior, however, by adding pre- or post-callbacks to any node. For example, you might integrate OpenGL into your application for a given node by using OpenGL in your DRAW callback functions.

Setting Up Node Callbacks

The following method enables you to set up pre- or post-callback functions for a node, with a given traversal that triggers the callback function.

```
void pfGetNodeTravFuncs(const pfNode* node, int which,
    pfNodeTravFuncType *pre, pfNodeTravFuncType *post);
```

node is the node for which the callback functions apply.

which is the kind of traversal that triggers the callback function. Possible values include:

- PFTRAV_APP
- PFTRAV_CULL
- PFTRAV_DRAW
- PFTRAV_ISECT

pre and *post* are pointers to callback functions triggered before or after, respectively, a node is traversed. Pre-callback functions completely replace the default behavior of the node; post-callbacks modify the default behavior of the node.

Use NULL as the value when not using a pre- or post-callback function.

You can use ***pfGetTravNode()** can be used inside the callback function. It returns the node for which a callback function was called.

Passing Data to Traversal Callback Functions

You can pass data to pre- or post-callback functions using the following function:

```
void pfNodeTravData(pfNode *node, int which, void *data);
```

node is the node for which the callback functions apply.

which is the kind of traversal that triggers the callback function.

data is a pointer to the data, allocated from shared memory, that is passed to the callback function.

Return Values for Traversal Callback Functions

The return value for your callback function must have one of three values:

- PFTRAV_CONT—continue with the traversal.
- PFTRAV_PRUNE—ignore the current node and its children but continue with the traversal.
- PFTRAV_TERM—terminate the traversal.

Sample Customized Traversals

You can create your own traversals to accomplish specific tasks, such as:

- Creating packed attribute arrays or GL display lists for objects in the scene graph.
- Finding the textures or nodes of a specific type in a scene graph.
- Computing bounding geometry.

OpenGL Performer includes in a pfuTraverser utility `libpfutil/trav.c` to help you write your own traverser. pfuTraverser recursively traverses the nodes in a scene graph database, applying pre- and post-traversal functions to each node.

Also included in `trav.c` is a series of general user traversers that implement pfuTraverser, as described in Table 6-3.

Table 6-3 General User Traversals

Traversal	Description
pfuTravPrintNodes	Prints the nodes encountered in a traversal.
pfuTravCountDB	Accumulates static graphics and database statistics for the tree under the given node.
pfuTravNodeHlight	Sets a given highlighting structure on all pfGeoSets under a given node.
pfuTravNodeAttrBind	Sets a given attribute to the given bind value on every pfGeoSet under the given node.
pfuTravCalcBBox	Computes the bounding box.
pfuTravCountNumVerts	Counts the number of vertices.
pfuTravSetDListMode	Sets the display-list pfGeoSet status.
pfuTravCreatePackedAttrs	Creates packed attributes.
pfuFillGSetPackedAttrs	Sets the values of packed attributes.
pfuDelGSetAttrs	Deletes attributes.
pfuTravCachedCull	Caches CULL stages.
pfuCalcDepth	Calculates the depth of the scene graph rooted at a node. A single root node with no children is counted as having a depth of one.

Table 6-3 General User Traversals **(continued)**

Traversal	Description
pfuLowestCommonAncestor	Finds the lowest common ancestor of all nodes under node for which a given function returns true.
pfuLowestCommonAncestorOfGeoSets	Finds the lowest common ancestor node of all GeoSets under node for which a given function returns true.
pfuFindTexture	Finds the nth texture under a given node for which a given function returns true.

Creating Geometry with pfGeoSets

OpenGL Performer provides pfGeoSets for holding low-level geometric descriptions of objects. A pfGeoSet is a collection of like-geometric primitives, such as points, line segments, triangle strips, triangles, or triangle fans. The primitives in a pfGeoSet share a state description for texture, material, and other surface attributes in a pfGeoState.

By combining multiple pfGeoSets you can create a complex object, such as a house, car, or terrain. By manipulating the vertices of pfGeoSet elements, you can create a dynamic object, such as ocean waves.

This chapter describes how to create geometric surfaces and place them in the scene graph in the following sections:

For more information about pfGeoState, see Chapter 8, "Specifying the Appearance of Geometry with pfState and pfGeoState."

Note: OpenGL Performer provides a richer alternative to the pfGeoSet class: the pfGeoArray class. To find out how it is used, see the *OpenGL Performer Programmer's Guide*.

---

pfGeoSet Overview

A pfGeoSet is a collection of one or more like primitives, such as lines or triangles. These primitives are arranged in a way that forms a geometric surface.

pfGeoSets contain:

- A defined primitive type set with **pfGSetPrimType()**.
- The function **pfGSetNumPrims()** specifies the number of primitives in the pfGeoSet.
- For stripped primitives, such as triangle strips, the number of vertices in each strip is set with a lengths array using **pfGSetPrimLengths()**.
- Vertex coordinate attribute lists, with optional corresponding index lists set with **pfGSetAttr()**.
- The kind of attribute binding, also specified in **pfGSetAttr()**, determines whether attributes are specified as follows:
  - Per vertex: PFGS_PER_VERTEX
  - Per primitive: PFGS_PER_PRIM
  - For all the primitives in the pfGeoSet: PFGS_OVERALL
- A reference to a pfGeoState specified with **pfGSetGState()**, which specifies the surface appearance (lighting material, texture, transparency, etc.) of the geometry.

A simple example of pfGeoSet creation and rendering demonstrating the concepts in this chapter can be found at `/usr/share/Performer/src/pguide/libpr/gset.c` on IRIX and Linux systems and at `%PFROOT%\Src\pguide\libpr\gset.c` on Windows systems.

For information about optimizing pfGeoSet performance, see “Optimizing pfGeoSet Performance” on page 221.

Creating a pfGeoSet

The following sections describe how to create a pfGeoSet.

Creating a pfGeoSet Object

To create a pfGeoSet from the shared memory arena, use the following line of code:

```
pfGeoSet *pfNewGSet(void *arena)
```

Setting the Primitive Type

Use the following pfGeoSet method to specify the type of primitive in the pfGeoSet:

```
void pfGSetPrimType(pfGeoSet *gset, int type);
```

type is one of the primitives provided by OpenGL Performer:

- PFGS_POINTS
- PFGS_LINES
- PFGS_LINESTRIPS
- PFGS_FLAT_LINESTRIPS
- PFGS_TRIS
- PFGS_QUADS
- PFGS_TRISTRIPS
- PFGS_FLAT_TRISTRIPS
- PFGS_TRIFANS
- PFGS_FLAT_TRIFANS
- PFGS_POLYS

PFGS_FLAT_* primitives are flat-shaded. PFGS_POLYS draws polygons of arbitrary vertex lengths.

Figure 7-1 shows some of these primitives.

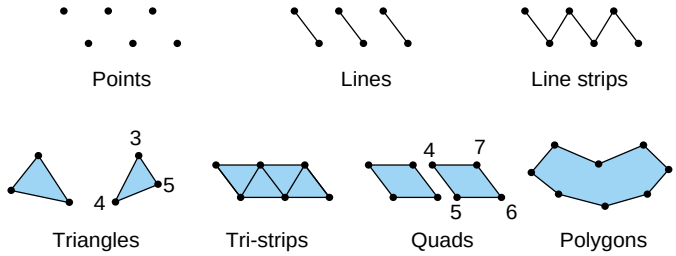

Figure 7-1 Primitives

The numbers in Figure 7-1 show the order in which the vertex attributes should appear in the attribute array.

Setting the Number of Primitives

Use the following pfGeoSet method to specify the number of primitives in the pfGeoSet:

```
void pfGSetNumPrims(pfGeoSet *gset, int num);
```

num is the number of primitives.

Setting the Number of Vertices Per Stripped Primitive

When using one of the following pfGeoSet primitives, which have an arbitrary number of vertices, you must define the number of vertices of each primitive in the pfGeoSet:

- PFGS_LINESTRIPS
- PFGS_FLAT_LINESTRIPS
- PFGS_TRISTRIPS
- PFGS_FLAT_TRISTRIPS
- PFGS_TRIFANS
- PFGS_FLAT_TRIFANS
- PFGS_POLYS

Use **pfGSetPrimLengths()** to specify the number of vertices of each primitive in the pfGeoSet:

```
void pfGSetPrimLengths(pfGeoSet* gset, int *lengths);
```

lengths is an array of the number of strips in a pfGeoSet. Each element of the *lengths* array is the number of vertices in a corresponding strip. For example:

```
lengths[0] = 8;
lengths[1] = 5;
```

These lines of code mean that the number 0 primitive has 8 vertices, and the number 1 primitive has 5 vertices. Use **pfGetGSetPrimLength()** to return the length of an individual primitive from the lengths array.

Note: pfGetGSetPrimLength() checks for NULL or negative lengths.

Attributes of pfGeoSet Primitives

The vertex information of the primitives in a pfGeoSet are described by attribute lists. Each element in an attribute list contains information for a single vertex. The vertex attributes for all vertices of all primitives of a pfGeoSet are stored in separate arrays, according to the following attribute types:

- Vertices—pfVec3 coordinates (required)
- Colors—pfVec4 colors
- Normals—pfVec3 normals
- Texture coordinates—pfVec2 texture coordinates

A pfGeoSet has at least one array pfVec3 of vertex coordinates. Optional attributes include colors, normals, and texture coordinates. Arrays holding these attributes for the vertices of the pfGeoSet are specified for the pfGeoSet using **pfGSetAttr()**. Figure 7-2 shows the arrays of attributes.

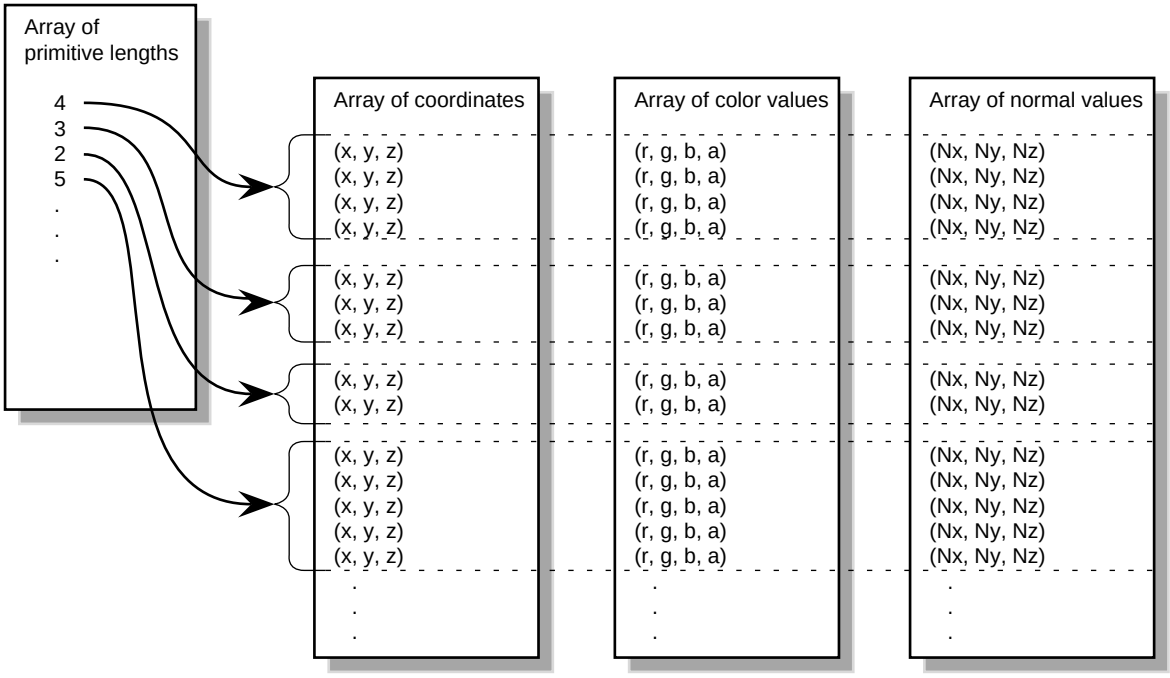

Figure 7-2 Arrays of Stripped Primitives

Indexes for indexed attributes are in separate index arrays. For more information about indexed attributes, see "Indexed Arrays" on page 118.

You can also put all vertex attributes of a pfGeoSet in a single-packed attribute array. You may use packed attribute arrays for performance reasons or for specifying specialized custom formats of data. For more information about packed attribute arrays, see "Packed Attributes" on page 120.

Setting the Attributes

To set the attributes of a pfGeoSet, use **pfGSetAttr()** as follows:

```
void setAttr(int attr, int bind, void *alist, ushort *ilist);
```

attr specifies the attribute array to set. The tokens for the different attribute lists are:

- PFGS_COLOR4
- PFGS_NORMAL3
- PFGS_TEXCOORD2
- PFGS_COORD3
- PFGS_PACKED_ATTRS

bind is the binding type. The tokens are listed in Table 7-1.

alist is a pointer to an attribute array for appropriate, corresponding data.

ilist is a pointer to an index array for used to access the attribute array.

Attribute Bindings

Attribute bindings specify whether attributes are as follows:

- Per vertex— PFGS_PER_VERTEX
- Per primitive—PFGS_PER_PRIM
- For all the primitives in the pfGeoSet—PFGS_OVERALL
- Unspecified—PFGS_OFF

For example, you can specify the following:

- A unique color for each vertex (PFGS_PER_VERTEX).
- A unique color for each primitive (PFGS_PER_PRIM).
- One color for all primitives in the pfGeoSet (PFGS_OVERALL).
- An unspecified color (PFGS_OFF).

Table 7-1 shows the possible bindings per attribute type.

Table 7-1 Possible Bindings Per Attribute Type

Binding Type	Colors	Normals	Texture Coordinates	Vertices
PFGS_OFF	Yes	Yes	Yes	No
PFGS_OVERALL	Yes	Yes	No	No
PFGS_PER_PRIM	Yes	Yes	No	No
PFGS_PER_VERTEX	Yes	Yes	Yes	Yes

Indexed Arrays

A cube has 6 sides; together those sides have 24 vertices. In a vertex array, you could specify the primitives in the cube using 24 vertices. However, most of those vertices overlap. If more than 1 primitive can refer to the same vertex, the number of vertices can be streamlined to 8. To get more than 1 primitive to refer to the same vertex, use an index; 3 vertices of 3 primitives use the same index, which points to the same vertex information. Adding the index array adds an extra step in the determination of the attribute, as shown in Figure 7-3.

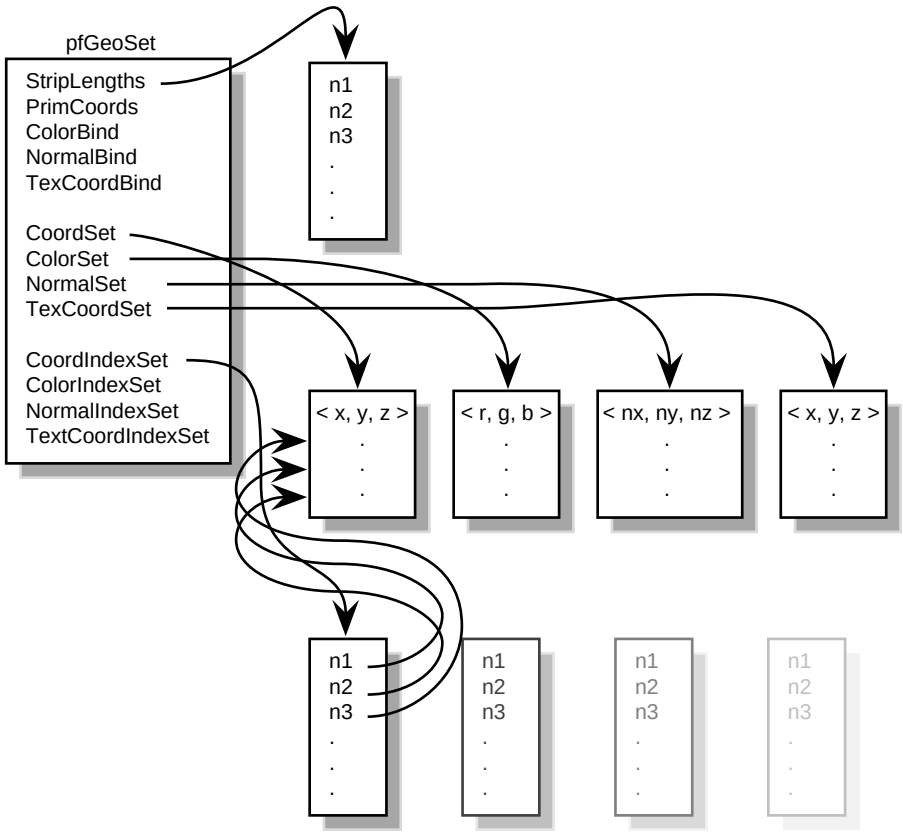

Figure 7-3 Indexing Arrays

Indexing can save system memory, but rendering performance is often lost.

Whether or not attributes should be indexed depends on how many vertices in a geometry are shared:

- If attributes are shared by many primitives, the attributes should be indexed.
- If attributes are not shared by many primitives, the attributes should be handled sequentially.

Consider the following two examples in Figure 7-4, in which each dot marks a vertex.

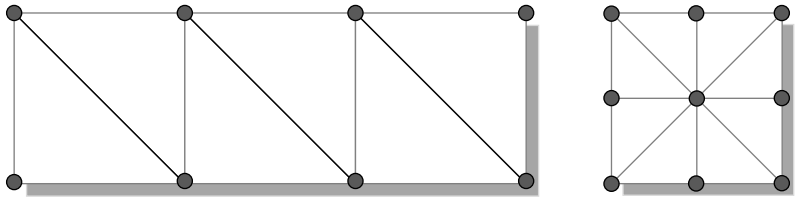

Figure 7-4 Deciding whether to Index Attributes

In the triangle strip, each vertex is shared by 2 adjoining triangles. In the square, the same vertex is shared by 8 triangles. Consider moving these vertices when, for example, morphing the object. If the vertices were not indexed in the square, the application would have to search for and alter 8 triangles to change one vertex. In the case of the square, it is much more efficient to index the attributes.

On the other hand, if the attributes in the triangle strip were indexed, because each vertex is shared by only 2 triangles, the index search time would exceed the time required to simply update the vertices sequentially. In the case of the triangle strip, rendering is improved by handling the attributes sequentially.

The choice of using indexed or sequential attributes applies to all of the primitives in a pfGeoSet. All of the primitives within one pfGeoSet must be referenced sequentially or by index; you cannot mix the two.

Packed Attributes

Using packed attributes is an optimized way of sending formatted data to the graphics pipeline under OpenGL operation. Using packed attributes can help host traversal performance because they remove subroutine call overhead. Packed attributes can also reduce memory usage because they allow for the format specification of attributes such as normals, texture coordinates as floats, and colors as unsigned bytes. Some small additional overhead might be incurred by the geometry subsystem of the graphics pipeline, which has to unpack the data.

The packed attribute array holds the currently bound per-vertex attribute data packed into a single non-indexed array and is specified with the matching format of the data with **pfGSetAttr()** as follows:

```
pfGSetAttr(gset, PFGS_PACKED_ATTRS, PFGS_PA_C4UBN3ST2F /*the format*/);
```

Vertex coordinate attributes can be placed in this array and do not need to be duplicated in their regular arrays. Specify NULL for the attribute list to **pfGSetAttr()**. Vertex coordinates themselves must always be provided in the normal vertex coordinate list. They can, based on the packed format, be duplicated in the packed array.

To create packed attributes, you can use the utility **pfuTravCreatePackedAttrs()**, which traverses a scene graph to create packed attributes according to the specified format for pfGeoSets and, optionally, pfDelete redundant attribute arrays. This utility packs the pfGeoSet attributes using **pfuFillGSetPackedAttrs()**. To then render geometry with packed attributes, use the **pfGSetDrawMode(PFGS_PACKED_ATTRS)** method when using OpenGL.

For more information on packed arrays on IRIX and Linux systems, see the following examples:

- `/usr/share/Performer/src/pguide/libpr/C/packedattrs.c`
- `/usr/share/Performer/src/sample/C/perfly.c`
- `/usr/share/Performer/src/sample/C++/perfly/perfly.C`

For more information on packed arrays on Windows systems, see following the examples:

- `%PFROOT%\Src\pguide\libpr\C\packedattrs.c`
- `%PFROOT%\Src\sample\C\perfly.c`
- `%PFROOT%\Src\sample\C++\perfly\perfly.C`

Also, see Chapter 8, “Geometry,” in the *OpenGL Performer Programmer’s Guide*.

Drawing and Printing a pfGeoSet

pfGeoSets are the lowest-level OpenGL Performer object that can be rendered. To directly draw a pfGeoSet, use **pfDrawGset()**.

pfGeoSets are pfObjects, so they can have their contents printed for debugging with different levels of verbosity via the pfObject routine, **pfPrint()**.You can also use other pfObject methods, such as **pfCopy()** and **pfDelete(),** can also be used with pfGeoSets.

Placing Geometry in a Scene Graph

You incorporate pfGeoSets into scene graphs using a pfGeode leaf node. A single pfGeode node can have multiple pfGeoSets associated with it if you use the **pfAddGSet()** method.

To place geometry in a scene graph, follow these steps:

1. Create a pfGeoSet with **pfNewGSet()**.
2. Attach the pfGeoSet to a pfGeoState using **pfGSetGstate()**.
3. Add the pfGeoSet to a pfGeode node in a scene graph using **pfAddGSet()**.

The result is shown in Figure 7-5.

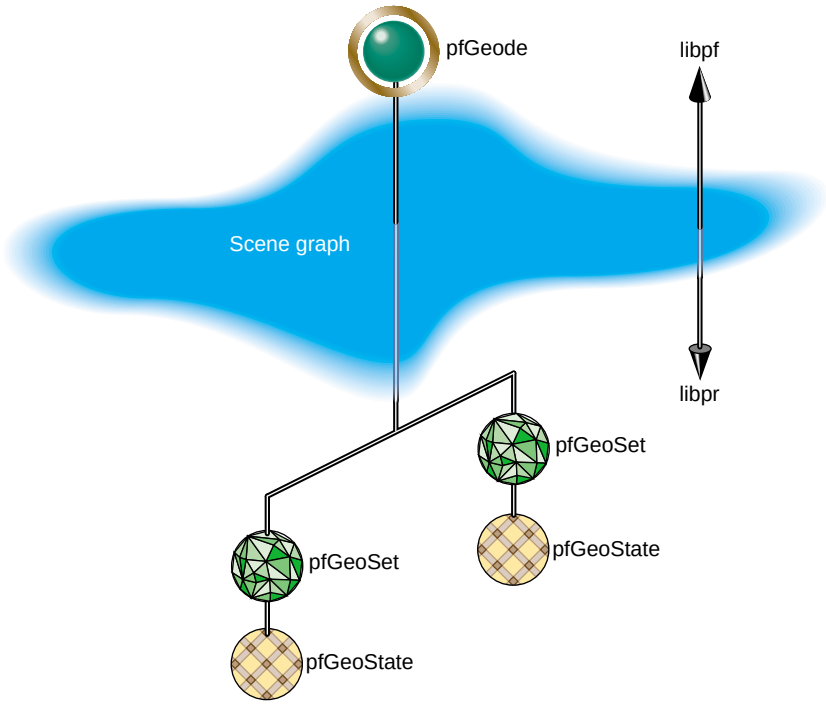

Figure 7-5 Geometry Objects

To create a pfGeode node, associate two pfGeoSets with it, and attach the node to the scene graph using code similar to the following:

```
pfGeode *geode = pfNewGeode();
pfAddSet( geode, gSet1 );
pfAddSet( geode, gSet2 );
pfAddChild( GeodesParentNode, geode );
```

The structure of a scene graph impacts the performance of your application. For more information, see "Arrangement of Nodes" on page 100.

Creating Common Geometric Objects

`libpfdu` provides routines to generate pfGeoSets for common shapes, including

- Sphere
- Cube
- Pyramid
- Cylinder
- Cone

When creating these shapes, you can specify the number of triangles that comprise them. For example, the following method sets the number of triangles comprising the sphere to 200:

```
pfGeoSet *sphere = pfdNewSphere(200, arena);
```

This method returns the number of vertices and normals in the shape.

The more triangles, the smoother the curves but the slower the rendering; fewer triangles allow faster rendering but produce more jagged curves.

Utilities to Create Common Geometric Objects

Table 7-2 shows the `libpf` utilities that generate pfGeoSets of larger geometric shapes.

Table 7-2 Common Geometric Objects

Geometry	Properties	Utilities That Create the Geometry
Sphere	Unit	extern pfGeoSet * pfdNewSphere(int ntris, void *arena)
	Radius=1, from Z=-1 to Z=1	extern pfGeoSet * pfdNewCylinder(int ntris, void *arena)
	Radius=1, from Z=0 to Z=1	extern pfGeoSet * pfdNewCone(int ntris, void *arena)
Cube	Unit	extern pfGeoSet * pfdNewCube(void *arena)
Pyramid	Unit square base, from Z=0 to Z=1	extern pfGeoSet * pfdNewPyramid (void *arena)
	Z=0 to Z=1	extern pfGeoSet * pfdNewArrow (int ntris, void *arena)
	Z=-1 to Z=1	extern pfGeoSet * pfdNewDoubleArrow (int ntris, void *arena)
Cylinder	Without end caps and variable radii	extern pfGeoSet * pfdNewPipe (float botRadius, float topRadius, int ntris, void *arena)
Circle	Unit circle facing +Z, filled	extern pfGeoSet * pfdNewCircle (int ntris, void *arena)
	Unit circle in Z=0 plane, lines	extern pfGeoSet * pfdNewRing (int ntris, void *arena);

Specifying the Appearance of Geometry with pfState and pfGeoState

A pfState holds the global graphic's state description. A pfGeoState encapsulates the graphics state elements, such as lighting, transparency, and texture that define the appearance of a pfGeoSet or pfGeoArray. (Hereafter, this section will refer only to pfGeosets.) Every pfGeoSet must reference a pfGeoState. State definitions for the pfGeoSet come either from its pfGeoState, or from the global, default settings in the global pfState.

This chapter describes how to define the appearance of geometries in the following sections:

Setting the Graphics State

Graphics state elements can be directly set in immediate mode through **pfApply*()** routines. For example, use **pfApplyMtl()** to set a current material; use **pfEnable()** to enable a specific mode, such as lighting, or use **pfApplyGState()** to set a complete collection of graphics state elements. When these calls are made, the current graphics state is recorded by OpenGL Performer in a pfState. This provides functionality, as well as optimizations, to prevent redundant graphics state changes.

Global State

pfState contains all global graphics state information and all of the information necessary to define the appearance of a geometry. You must have a current pfState to create any

other OpenGL Performer state objects or to do any graphics operations; however, pfWindows by default automatically creates and selects its own pfState when it is opened. You can create and select a pfState object with **pfInitState()**. There is also a state stack that can be pushed and popped in pfState with **pfPushState()** and **pfPopState()**. You can lock state settings can be locked by using the pfState function **pfOverride()** to prevent future changes. pfState graphic state values become the default appearance values for all pfGeoSets.

pfGeoStates state values found in pfState and are primarily used for specifying the appearance of geometry. pfGeoStates can specify the following, among other things:

- Material properties with the pfMaterial state attribute object
- Textures with the pfTexture state attribute object
- Transparency with the transparency mode

When a pfGeoState is created, it is configured to inherit all appearance values from the current pfState when it is applied. Those values can be changed using methods in pfGeoState: **pfGStateAttr()**, **pfGStateMode()**, and **pfGStateVal()**. pfGeoStates settings can be applied directly to the current global state with **pfLoadGState()**. pfGeoStates referenced by pfGeoSets only affect the pfGeoSets referencing them. **pfApplyGState()** will set state values for subsequent pfGeoSets, but those state values will revert back to the previous pfState values for the next call to **pfApplyGState()**. A pfGeoState can be directly loaded into the current pfState to set inherited values by future pfGeoStates with **pfLoadGState()**.

Defining a pfGeoState

To define a pfGeoState, follow these steps:

1. Create a pfGeoState object using **pfNewGState()**.
2. Associate the pfGeoState appearance values with a geometry using **pfGSetGState()**.
3. Specify the modal graphic states, such as enables, you want to change using **pfGStateMode()**.
4. Specify the attribute graphic states you want to change, such as textures and materials, using **pfGStateAttr()**.

For example, to enable lighting and antialiasing and to set the material of the geometry to metal, use code similar to the following:

```
pfMaterial *mtl = pfNewMtl(arena);

pfGeoState *gstate = pfNewGState(arena);
pfGStateMode(gstate, PFSTATE_ENLIGHTING, PF_ON);
pfGStateMode(gstate, PFSTATE_ANTIALIAS, PFAA_ON);
pfGStateAttr(gstate, PFSTATE_FRONTMTL, mtl);
```

Setting pfGeoState Values for a Scene

Generally, pfGeoState values alter global, pfState appearance values for specific pfGeoSets. You can also use a pfGeoState to set global state values by attaching a pfGeoState object to a scene node using **pfSceneGState()**, as follows:

```
void pfSceneGState(pfScene *scene, pfGeoState *gstate);
```

OpenGL Performer does a **pfPushState()** and a **pfLoadGState()** of the pfScene pfGeoState before rendering the scene graph.

pfGeoStates and pfGeoSets

pfState contains the default, global state values, many of which define the appearance of the geometric objects in the scene. When pfGeoSets are drawn with **pfDrawGSet()**, they automatically call **pfApplyGState()** on their pfGeoState. If a pfGeoState is not defined for a geometry, the appearance values are undefined. To inherit all values from the global pfState, a pfGeoSet should have a pfGeoState with all values set to inherit, which is the default. A state value defined for a specific pfGeoSet takes precedence over the corresponding global state value.

For an example of a pfGeoState used globally, see “Computing the Optimal, Global Graphics State” on page 222.

Optimizing Graphics State Changes

Changing the graphics context from the global value to a value defined for a specific geometry impacts the performance of an application. For that reason, it is important to set the global appearance values to satisfy most geometries, thus changing the local appearance values as little as possible.

For more information about optimizing graphic state changes, see “Optimizing Graphics State Changes” on page 222.

Setting Modal pfGeoState Values

Many pfGeoState graphic states, such as transparency, are specified with a token. These graphic states are set using **pfGStateMode()**.

Table 8-1 shows the modal graphic states you can specify along with their possible values and defaults.

Table 8-1 Graphic States

Graphic State	Possible Values	Default Value
PFSTATE_TRANSPARENCY	See “pfTransparency” on page 129	PFTR_OFF
PFSTATE_ANTIALIAS	PFAA_OFF, PFAA_ON	PFAA_OFF
PFSTATE_DECAL	See “pfDecal” on page 129	PFDECAL_OFF
PFSTATE_ALPHAFUNC	See “pfAlphaFunc” on page 130	PFAF_ALWAYS
PFSTATE_ALPHAREF	Float between 0.0 and 1.0	0.0
PFSTATE_ENLIGHTING	PF_OFF, PF_ON	PF_OFF
PFSTATE_ENTEXTURE	PF_OFF, PF_ON	PF_OFF
PFSTATE_ENFOG	PF_OFF, PF_ON	PF_OFF
PFSTATE_CULLFACE	PFCF_OFF, PFCF_BACK, PFCF_FRONT, PFCF_BOTH	PFCF_OFF
PFSTATE_ENWIREFRAME	PF_OFF, PF_ON	PF_OFF
PFSTATE_ENCOLORTABLE	PF_OFF, PF_ON	PF_OFF
PFSTATE_ENHIGHLIGHTING	PF_OFF, PF_ON	PF_OFF
PFSTATE_ENLPOINTSTATE	PF_OFF, PF_ON	PF_OFF
PFSTATE_ENTEXGEN	PF_OFF, PFTG_OBJECT_LINEAR, PFTG_EYE_LINEAR, PFTG_EYE_LINEAR_IDENT	PF_OFF
PFSTATE_ENTEXLOD	PF_OFF, PF_ON	PF_OFF
PFSTATE_ENTEXMAT	PF_OFF, PF_ON	PF_OFF

pfTransparency

pfTransparency sets the type of transparency computation used for rendering transparency effects. The different types of transparency computations define how the geometry's color and the framebuffer color are blended. Transparency can have different performance and image-quality characteristics on different graphics subsystems. For this reason, it is better to provide OpenGL Performer with a hint, such as PFTR_HIGH_QUALITY, rather than specifying a method that does not work on all platforms; OpenGL Performer interprets the hint, PFTR_HIGH_QUALITY, for all platforms.

Transparency modes include:

- PFTR_OFF—the default, draws transparent objects as opaque.
- PFTR_ON—allows OpenGL Performer to choose the default mode based on speed and quality.
- PFTR_HIGH_QUALITY—uses methods for highest image quality.
- PFTR_FAST—uses methods for fastest rendering.
- PFTR_BLEND_ALPHA—OpenGL glBlendFunc(3g) method.
- PFTR_MS_ALPHA—OpenGL glEnable(GL_SAMPLE_ALPHA_TO_ONE_SGIS) method when multisampling is available and enabled.
- PFTR_MS_ALPHA_MASK—OpenGL glEnable(GL_SAMPLE_ALPHA_TO_MASK_SGIS) when multisampling is enabled.

pfDecal

Decaled geometry can be thought of as a stack, where each layer has visual priority over the geometry beneath it in the stack. As with transparencies, different hardware platforms offer different methods with different performance and image quality characteristics. For this reason, OpenGL Performer allows and recommends that unless you have specific motivation, use a hint rather than a specific method, which might not work on all platforms.

pfDecal modes include:

- PFDECAL_OFF
- PFDECAL_BASE

- PFDECAL_LAYER
- PFDECAL_BASE_FAST, PFDECAL_LAYER_FAST
- PFDECAL_BASE_HIGH_QUALITY, PFDECAL_LAYER_HIGH_QUALITY
- PFDECAL_BASE_DISPLACE, PFDECAL_LAYER_DISPLACE
- PFDECAL_BASE_DISPLACE, PFDECAL_LAYER_DISPLACE_AWAY
- PFDECAL_BASE_STENCIL, PFDECAL_LAYER_STENCIL
- PFDECAL_PLANE

See the `pfDecal` man page for the definition of the decal mode values. PFDECAL_OFF is the default.

pfAlphaFunc

pfAlphaFunc sets the alpha function mode. The alpha function mode specifies whether or not a given pixel is rendered according to its alpha value. For example, if you set pfAlphaFunc to PFAF_GREATER, only pixels with alpha values greater than a reference value are rendered.

You specify the reference value using PFSTATE_ALPHAREF. For example, to render only those pixels with alpha values greater than 0.5, use the following code:

```
pfGStateMode(gstate, PFSTATE_ALPHAFUNC, PFAF_GREATER);
pfGStateValue(gstate, PFSTATE_ALPHAREF, 0.5);
```

Alpha Func Modes

PFSTATE_ALPHAFUNC is the function you use to compare a reference alpha value with the alpha value of a geometry. PFSTATE_ALPHAFUNC must be set to one of the following modes:

- PFAF_ALWAYS
- PFAF_EQUAL
- PFAF_GEQUAL
- PFAF_GREATER
- PFAF_LEQUAL
- PFAF_LESS

- PFAF_NEVER
- PFAF_NOTEQUAL
- PFAF_OFF

See the `pfAlphaFunc` man page for the definition of the PFSTATE_ALPHAFUNC mode values.

Setting pfGeoState Attributes

Many pfGeoState graphic states are specified using an object, such as pfMaterial. These graphic states are set using **pfGStateAttr()**. To use an object as the definition for an attribute, you must create the object and define it before calling **pfGStateAttr()**.

Table 8-2 shows the attribute pfGeoState values and the objects that define them.

Table 8-2 Attribute pfGeoState Values

Attribute	Object
PFSTATE_FRONTMTL	pfMaterial
PFSTATE_BACKMTL	pfMaterial
PFSTATE_TEXTURE	pfTexture
PFSTATE_TEXENV	pfTexEnv
PFSTATE_FOG	pfFog
PFSTATE_LIGHTMODEL	pfLightModel
PFSTATE_LIGHTS	pfLight
PFSTATE_COLORTABLE	pfColortable
PFSTATE_HIGHLIGHT	pfHighlight
PFSTATE_LPOINTSTATE	pfLPointState
PFSTATE_TEXGEN	pfTexGen
PFSTATE_TEXLOD	pfTexLOD

Table 8-2 Attribute pfGeoState Values **(continued)**

Attribute	Object
PFSTATE_TEXMAT	pfMatrix
PFSTATE_DECALPLANE	pfPlane

All attributes default to NULL, which means that OpenGL default values are used.

For more information about any of the attributes, see the man pages of the objects associated with them.

Using Textures

Textures are images that are applied to the surface of a geometry, as shown in Figure 8-1.

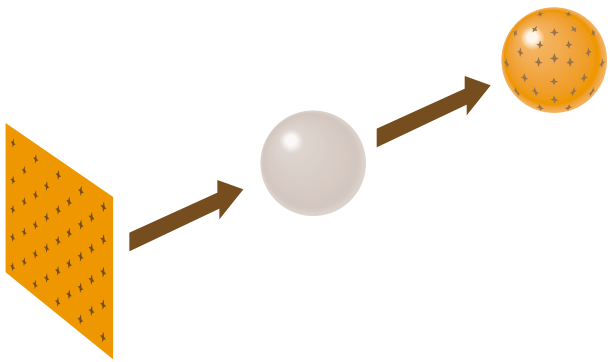

Figure 8-1 Applying Textures to Geometries

Textures can add tremendous realism to the rendered scene because they can be real photographs. An image of the pitted rind of an orange applied to a sphere, for example, creates a realistic-looking orange.

To use a texture, follow these steps:

1. Enable texture mapping.
2. Create a pfTexture.

3. Load or create the texture image and assign it to the pfTexture.
4. Optionally set the texture environment using pfTexEnv.
5. Set the texture coordinates on the pfGeoSet using pfGSetAttr, or else use a pfTexGen in the pfGeoState to automatically generate texture coordinates.

Enabling Texture Mapping

Texture mapping is expensive; consequently, by default, it is turned off. Enable texture mapping only for those objects that are textured.

To enable texture mapping for a geometry, use the following pfGeoState methods:

```
pfGStateMode(gstate, PFSTATE_ENTEXTURE, PF_ON);
pfGSetGState(gset, gstate);
```

The first line enables texture mapping, and the second line selects the pfGeoSet that is to be texture mapped.

Creating a Texture Object

To create a pfTexture, use the following method:

```
pfTexture *pfNewTex(void *arena)
```

The *arena* is that part of memory shared by all OpenGL Performer processes.

Loading an Image as a Texture

The easy way to set up a pfTexture is to load an image file. To load an image and make it a texture, use **pfLoadTexFile()**:

```
int pfLoadTexFile(pfTexture *tex, char *fname)
```

The texture must be in either the SGI format or the fast-loading OpenGL Performer PFI format. The texture is created with reasonable defaults for the specified image for the various control modes discussed further in this section.

You can set the paths that OpenGL Performer uses to find the image file *fname* using **pfFilePathv()**, for example:

```
pfFilePathv(“/usr/demos/data/textures",
           "/usr/demos/data/images",
           "/usr/share/Performer/data”,
           NULL);
```

OpenGL Performer searches through the directories in the order of their specification.

Preloading Textures

Downloading texture images from disk to the arena is time consuming. You can improve the performance of your application if you download all of the textures that your application needs one time.

Two tools help you preload textures:

```
pfList *pfuMakeSceneTexList(pfScene *scene)
void pfuDownloadTexList(pfList *list, int mode)
```

pfuMakeSceneTexList() traverses the scene graph and builds a list of all the textures used. **pfuDownloadTexList()** downloads the textures specified in the list to the GL and hardware texture memory and must be called from the DRAW process.

Specifying Texture Attribute

The texture image can be loaded or generated by some other utility besides **pfLoadTexFile()**, and in this case you must fully specify the texture image details with **pfTexImage()**.

```
void pfTexImage(pfTexture* tex, uint* image,
       int comp, int sx, int sy, int sz);
```

Textures can have as many as four components. The following are example uses:

- One component—consisting of intensity (I) or luminance (L) only, useful geometries that repeat but vary in contrast, such as grass and sand.
- Two components—consisting of intensity and transparency (IA), useful for geometries that repeat but vary in contrast and transparency, such as clouds.
- Three components—consisting of red, green, and blue (RGB).
- Four components—consisting of red, green, blue, and alpha (RGBA), useful for full-color textures.

A texture object also contains information about the handling of the image data, including

- Image data formats: host memory external format, internal hardware format, and the type of image data (RGB, Luminance, Intensity, etc.), set with **pfTexFormat()**.
- Minification or magnification filters, which specify whether the image is reduced or magnified before being applied to the surface of a geometry, set with **pfTexFilter()**.

  Texture wrap options, which specifies what happens when the texture is too small to completely cover a geometry, set with **pfTexRepeat()**. Options include repeating the texture until the geometry is covered or expanding the texture so that it covers the geometry.

The prototypes for these basic configuration routines are:

```
void pfTexFormat(pfTexture *tex, int format, int type);
void pfTexFilter(pfTexture *tex, int filt, int type);
void pfTexRepeat(pfTexture *tex, int wrap, int type);
```

Specifying Texture Formats

The format in which an image is stored in texture memory is defined with **pfTexFormat()**:

```
void pfTexFormat(pfTexture *tex, int format, int type)
```

format specifies which format to set. Valid formats and their basic types include:

- PFTEX_INTERNAL_FORMAT— specifies how many bits per component are to be used in internal hardware texture memory storage. The default is 16-bits per full texel and is based on the number of components and external format.
- PFTEX_IMAGE_FORMAT— describes the type of image data and must match the number of components, such as PFTEX_LUMINANCE, PFTEX_LUMINANCE_ALPHA, PFTEX_RGB, and PFTEX_RGBA. The default is the token in this list that matches the number of components. Other OpenGL selections can be specified with the GL token.
- PFTEX_EXTERNAL_FORMAT—specifies the format of the data in the pfTexImage array. The default is packed with 8-bits per component. There are special fast-loading hardware ready formats, such as PFTEX_UNSIGNED_SHORT_5_5_5_1.
- PFTEX_SUBLOAD_FORMAT—a boolean to specify if the texture will be a sub-loadable paging texture. Default is FALSE.

In general, you just need to specify the number of components in pfTexImage(). You may want to specify a fast-loading hardware-ready external format, such as PFTEX_UNSIGNED_SHORT_5_5_5_1, in which case OpenGL Performer will automatically choose a matching internal format. See the `pfTexFormat(3pf)` man page for more information on texture configuration details.

Setting the Texture Environment

The environment specifies how the potentially lit colors of the geometry and the texture image interact. This is described with a pfTexEnv object. The mode of interaction is set with **pfTEnvMode()**, and valid modes include:

PFTE_MODULATE—gray scale of the geometry is mixed with the color of the texture (the default). This option multiplies the shaded color of the geometry by the texture color. If the texture has an alpha component, the alpha value modulates the geometry's transparency. For example, if a black and white texture, such as text, is applied to a green polygon, the polygon remains green and the writing appears as dark green lettering.

- PFTE_DECAL—texture alpha component acts as a selector between 1.0 for the texture color, and 0.0 for the base color, to decal an image onto geometry.
- PFTE_BLEND—alpha acts as a selector between 0.0 for the base color and 1.0 for the texture color modulated by a constant texture blend color specified with **pfTEnvBlendColor()**. The alpha/intensity components are multiplied.
- PFTE_ADD—RGB components of the base color are added to the product of the texture color modulated by the current texture environment blend color. The alpha/intensity components are multiplied.

Setting the Texture Coordinates

The texture coordinates specify how the coordinates of the texture map to the coordinates of the geometry, as shown in Figure 8-2.

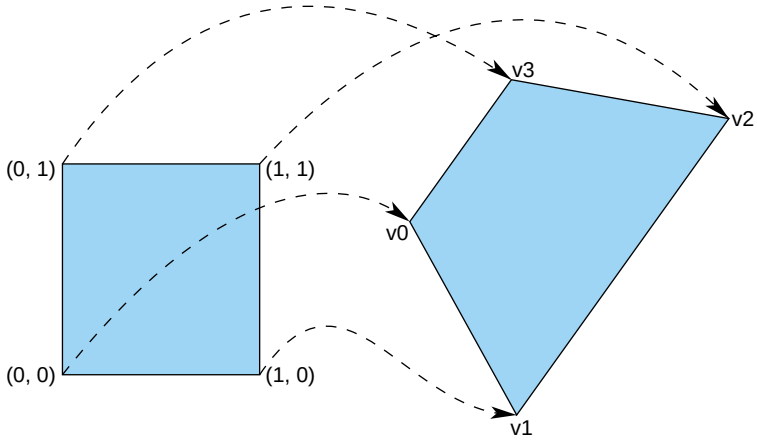

Figure 8-2 Texture Coordinates

The pfGeoSet method **pfGSetAttr()** specifies the texture coordinates for mapping each vertex of a pfGeoSet into texture space:

```
void pfGSetAttr(pfGeoSet *gset, PFGS_TEXCOORD2, PFGS_PER_VERTEX,
    void *alist, ushort *llist)
```

PFGS_TEXCOORD2 specifies pfVec2 texture coordinates.

PFGS_PER_VERTEX means the attribute is specified once per vertex.

alist is a pointer to the array of pfVec2 texture coordinates.

ilist is an optional pointer to the indices in the texture coordinate array.

Texture coordinates can also be automatically generated by various functions specified by a pfTexGen object. See Chapter 9, “Graphics State,” in the *OpenGL Performer Programmer’s Guide*, and the `pfTGenMode(3pf)` man page for more information on this object.

Specifying the Material

A material specifies the color of a geometry under different lighting conditions and opacity. There are five lighting conditions:

- Specular—highlights, such as shiny glints, (0.0, 0.0, 0.0), by default.
- Diffuse—directly illuminated portions of the geometry outside the specular region (0.8, 0.8, 0.8), by default.
- Ambient—those portions of the geometry illuminated by background lighting (0.2, 0.2, 0.2), by default.
- Emissive—color of the light emanating from the shape (0.0, 0.0, 0.0), by default.
- Alpha—transparency of the texture; the default, 1.0, is completely opaque.

Figure 8-3 shows three of these lighting conditions on a sphere illuminated from above.

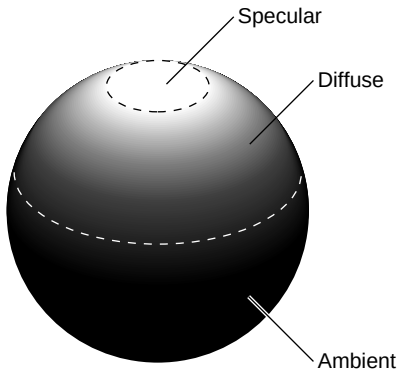

Figure 8-3 Light Characteristics

Specifying the Color and Shininess

To create a material and specify its color and shininess, use the following methods:

```
pfMaterial *   pfNewMtl(void *arena);
void pfMtlColor(pfMaterial *mtl, int color, float r, float g, float b);
void pfMtlShininess(pfMaterial *mtl, float shininess);
```

arena is memory allocated from the shared memory arena.

color specifies one of the lighting conditions:

- PFMTL_AMBIENT
- PFMTL_DIFFUSE

- PFMTL_SPECULAR
- PFMTL_EMISSION

To define more than one of these lighting conditions, use the method repeatedly with a different token for color each time.

shininess is a float between 0.0 and 128.0 where 0.0 is very dull and 128.0 is very shiny.

Color Mode

Loading material information is computationally intensive. In some situations, you can take a shortcut. For example, consider the case where you have three differently colored but otherwise identical balls. Rather than reload a new material for each ball, you can change the color of the material of each ball through the object colors. **pfMtlColorMode()** specifies the particular material attribute that can be set through object or vertex colors. Changing a material color this way is much faster than switching to a different material; it allows for sharing of materials and shading control.

The default is PFMTL_CMODE_AD, which sets the material's ambient and diffuse colors with the pfGeoSet colors. To turn this default functionality off, set the color mode to PFMTL_CMODE_COLOR, so that geometry colors will only set the current GL color and will not affect the material state.

Material Side

With the method **pfMtlSide()**, you can specify whether to apply the material on the side facing the viewer (PFMTL_FRONT), the side not facing the viewer (PFMTL_BACK), or both (PFMTL_BOTH). Back-sided lighting only takes effect if a two-sided lighting model is active. Two-sided lighting typically has a significant performance cost.

Object materials only have effects when lighting is active.

Specifying Lighting

Lighting requires a specified lighting model, an active light, and the enabling of graphics lighting operations. As lighting is typically applied to an entire scene, you probably want to enable lighting in your global state with **pfEnable(PFEN_LIGHTING)**, or in the scene pfGeoState:

```
pfGeoState *gstate = pfNewGState(arena);
pfScene *sceneNode = pfNewScene(void);

pfGStateMode(gstate, PFSTATE_ENLIGHTING, PF_ON);
pfSceneGState(sceneNode, gstate);
```

The lighting model, specified with the pfLightModel state attribute object, describes the type of lighting operations to be considered, including local lighting, two-sided lighting, and light attenuation. The fastest light model is infinite single-sided lighting. A light model also allows you to specify ambient light for the scene, such as might come from the sun, with **pfLModelAmbient()**.

You create pfLights by calling **pfNewLight()**. Lights have color and position. The light colors are specified with **pfLightColor()**:

```
void pfLightColor(pfLightSource* lsource, int which, float r,
   float g, float b);
```

which specifies one of three light regions:

- PFLT_AMBIENT
- PFLT_DIFFUSE
- PFLT_SPECULAR

r, *g*, and *b* specify the color components of the specified light color.

To position the light source using pfLightPos():

```
void pfLightPos(pfLight* light, float x, float y,
    float z, float w);
```

w is the distance between the location in the scene defined by (*x*, *y*, *z*) and the light source, *lsource*. If *w* equals zero, *lsource* is infinitely far away and (*x*, *y*, *z*) defines a vector pointing from the origin in the direction of *lsource*. If *w* equals one, *lsource* is located at the position, (*x*, *y*, *z*). The default position is (0, 0, 1, 0): directly overhead, infinitely far away.

pfLights are attached to a pfGeoState through the PFSTATE_LIGHTS attribute.

For moving lights in a *libpf* scene, you can use a pfLightSource node. pfLightSource defines a pfLight with light color and position. They take effect when lighting is active.

pfLightSource nodes are nodes that can be placed in the scene graph and have their position transformed by pfSCS and other transform nodes. pfLightSource nodes are active for the rendering of the entire scene. pfLightSource nodes are not pfLights and cannot be attached to pfGeoStates, and visa vera.

pfLights cannot be attached directly to the scene graph and must be attached to a pfGeoState.

Placing Geometry in a Scene

When you create a geometry, it has a specified size, location, and orientation, as defined in its own space. You can place such a geometry:

- In relationship to other shapes in the same scene.
- Into the coordinate system of the root node, known as world space.

This chapter describes how to perform these tasks in the following sections:

World Space and Object Space

Geometries are often created in a local coordinate system, modeled at the origin. To place geometries in a scene, the geometry must be given positions in the scene, or world space. This transformation of location establishes a new local transformed coordinate system. OpenGL Performer allows you to specify these transformations in the scene graph to position geometries, as shown in Figure 9-1.

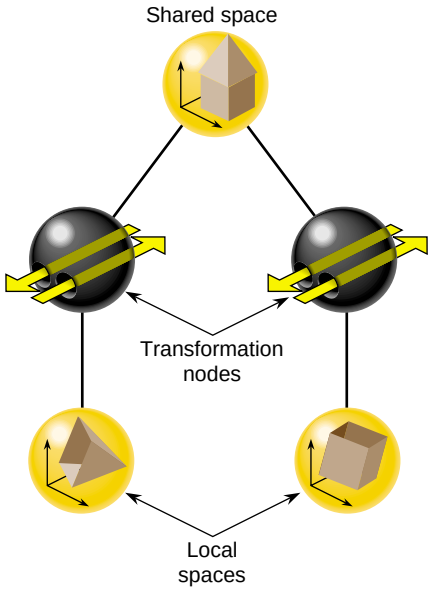

Figure 9-1 Shared Space

Transformation Node Isolation

Because a scene graph can be very wide and very deep, containing thousands of shapes, transformation nodes are often stacked. The transformation node at the top of the branch concatenates the transformations of all the transformation nodes directly below it. Transformations, however, are not carried over from one branch to another. For example, in Figure 9-1, the transformation node in the left branch does not affect the shape in the right branch.

As transformation nodes are encountered in the traversal, they are post-multiplied: (Geometry x TransformB) x TransformA.

World Space

When you want to put all of the shapes in a scene graph into one space, you use multiple transformation nodes to translate the shapes into the coordinate system of the root node of the scene graph. The coordinate system of the root node is called *world space*.

Geometry space is the coordinate system in a subsection of a scene graph.

Transformation Nodes

To put shapes together in a common space, or to reorient, reposition, or rescale a shape, you use one of two transformation nodes:

- pfFCS—for dynamically transforming geometries, in concert with pfFlux, to create movement.
- pfDCS—for transformation values that do change.
- pfSCS—for transformation values that do not change once they are set.

If you have shapes, like a rock, that do not move in a scene, use a pfSCS node to transform them. If a shape, such as a car, does move in a scene, use a pfDCS node to transform it.

Tip: Because pfDCS nodes require more processing, make as few pfDCS nodes as possible.

Transformation Node Functionality

Each of the transformation nodes provide methods to scale, rotate, or translate a shape to the coordinate system of the transformation's parent node.

For a given transformation node, multiple transformations are applied in the following order: scale, rotate, translate.

Ordering Transformation Nodes in the Scene Graph

The order in which you perform transformations can affect the final result. Consider, for example, translating and rotating an image. If you perform the transformations in this order, you end up with a rotated model translated, for example, down the X-axis, as shown in Figure 9-2.

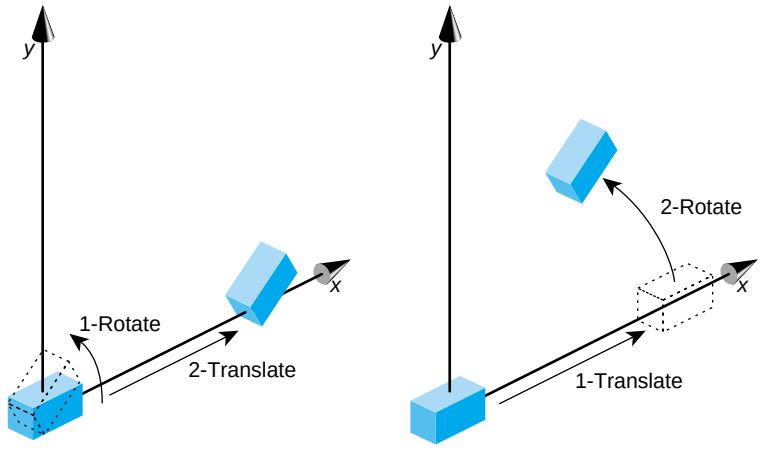

Figure 9-2 Order of Transformations

When you reverse the order of the transformations, the end result is different. Because the center of rotation is about the origin, the rotation transformation lifts the object above the X-axis.

Using pfFCS

pfFCS is used as a parent node to a pfGeode, containing pfGeoSets or pfGeoArrays. pfFCS can place the transformation matrix in a pfFlux object. pfFlux is a container for holding dynamic data, and stores the output data of a pfEngine. A pfEngine then can update the transformation matrix held in the pfFlux object, which transforms the child node of the pfFCS, as shown in Figure 9-3.

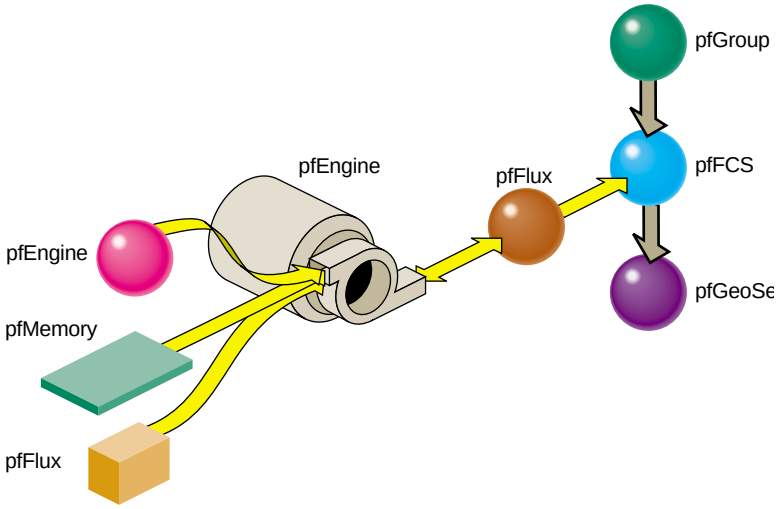

Figure 9-3 pfEngine Drives a pfFlux Node Animating a pfFCS Node

In this figure, the pfEngine performs calculations on the data input from the pfMemory nodes and sends the results to the pfFlux node. The pfFlux node contains the matrix for the pfFCS node. The output data from the pfEngine, manipulates the matrix in pfFlux, which, in turn, manipulates the pfGeode geometry, which is wrapped in the pfFCS node. In this way, the pfEngine animates the pfGeode geometry.

For more information about pfFlux, pfEngine, and pfFCS nodes, see Chapter 14, "Dynamic Data," in the *OpenGL Performer Programming Guide*.

pfFCS, pfFlux, and pfEngine Example

Example 9-1 shows an implementation of pfFCS, pfFlux, and pfEngine.

Example 9-1 Connecting Engines and Fluxes

```
// create the nodes
pfFlux *myData1 = new pfFlux(100 * sizeof(pfVec3));
pfFlux *myData2 = new pfFlux(100 * sizeof(pfVec3));
pfEngine *myEngine = new pfEngine(PFENG_SUM);
pfFlux *engineOutput = new pfFlux(100 * sizeof(pfVec3));
pfFCS myFCS = new pfFCS();
```

```
pfGeode myGeode = new pfGeode();

// initialize and populate the flux nodes
myData1->init();
myData2->init();

// attach the pfFlux nodes as the source of the pfEngine
myEngine->setSrc(0, myData1, 0, 3);
myEngine->setSrc(0, myData12, 0, 3);

// attach a pfFlux to the output of the pfEngine
myEngine->setDst(engineOutput, 0, 3);
myEngine->iterations(100, 3);

// connect the pfFlux output node to the scenegraph
myFCS->setFlux(engineOutput);
// attach child geometry to be tranformed by the FCS
myFCS->addChild(myGeode);

...
// compute the data in the source pfFluxes to the engine
float *current = (float *)myData1->getWritableData();
... // compute data
myData1->writeComplete();
```

Using DCS Nodes

You use DCS nodes to transform shapes when the transformations might change over time. For example, a rotating wheel changes its rotational angle over time.

Creating a DCS Node

To create a DCS node, use the following member function:

```
pfDCS *pfNewDCS(void);
```

Setting the DCS Node

To set the orientation, rotation, and scaling of the shape in the transformation's parent node, use the following methods, respectively:

```
void pfDCSTrans(pfDCS *dcs, float x, float y, float z);
void pfDCSRot(pfDCS *dcs, float h, float p, float r);
void pfDCSScale(pfDCS *dcs, float s);
```

pfDCS includes overwritten forms of these methods so that you can express the arguments in different units.

pfDCSScale() scales all three axes the same amount.

All of the transformation values only take effect when the DCS node is traversed by a DRAW action.

Using pfDCSCoord

An alternative to specifying translation and rotation values separately is using the **pfDCSCoord()** method, as follows:

```
void pfDCSCoord(pfDCS *dcs, pfCoord *coord)
```

This call is equivalent to:

```
pfDCSTrans(dcs, coord.xyz[PF_X], coord.xyz[PF_Y], coord.xyz[PF_Z]);
pfDCSRot(dcs, coord.hpr[PF_H], coord.hpr[PF_P], coord.hpr[PF_R]);
```

Optimizing the Use of DCS Nodes

By default, OpenGL Performer recalculates bounding volumes every time a transformation node is updated. To reduce the number of times a bounding volume is recalculated requires special knowledge of your visualization. For example, if your visualization is that of a solar system, every time a planet moves around the sun, its bounding volume is recalculated. Instead, by knowing the dimensions of your solar system model, you can set the bounding volume large enough so that it encompasses the motion of the planet and therefore never needs to be recalculated. Set the bounding box as high in the scene graph as you can at the transform node itself.

To turn off bounding volume recalculation, use the PFBOUND_STATIC token as the value for *mode* in the following **pfNode** method:

```
void pfNodeBSphere(pfNode *node, pfSphere *sph, int mode);
```

node, in this case, is the DCS node.

sph is the bounding sphere whose size you set so that recalculating the bounding sphere is unnecessary. If you set *sph* to NULL, the bounding sphere is automatically calculated.

OpenGL Performer makes internal optimizations based on knowing the matrix type, and based on calls, such as **pfDCSTrans()** and **pfDCSRot()**. Otherwise you can specify the matrix type using **pfDCSMatType()**.

Using SCS Nodes

An SCS node contains a transformation matrix that concatenates matrices for translating, rotating, and scaling a shape. The value of using a matrix is its speed of computation. The transformation values in a SCS node, however, cannot be changed once the SCS node is created.

Creating a SCS Node

To create an SCS node, use the following member function:

```
pfSCS *pfNewSCS(pfMatrix mat);
```

mat is the concatenation of the matrices for translating, rotating, and scaling a shape.

Setting the SCS Node

To set the transformation values in a SCS node:

1. Use **pfMatrix pfMake..Mat()** methods to define the first transformation matrix value.
2. Use the resulting matrix as the argument for creating the SCS node.
3. Use **pfMatrix pfPre..Mat()** methods to define the remaining transformation matrix values.

Note: Alternatively, you can use only **pfMatrix pfMake..Mat()** methods to define the transformations and then use **pfMultMat()** iteratively to multiply the three transformation matrices to yield the argument for **pfNewSCS()**.

Setting the First Transformation Matrix

To set the orientation, rotation, and the scaling transformation values, use the following pfMatrix methods, respectively:

```
void pfMakeTransMat(pfMatrix dst, float x, float y, float z);
void pfMakeRotMat(pfMatrix dst, float degrees, float x, float y,
    float z);
void pfMakeScaleMat(pfMatrix dst, float x, float y, float z);
```

dst in each method is the output transformation matrix for the function.

Setting the Remaining Transformation Matrices

You can concatenate transformation matrices by setting up one transformation method and then using one of the following pfMatrix methods:

```
void   pfPreTransMat(pfMatrix dst, float x, float y, float z,
    pfMatrix m);
void   pfPreRotMat(pfMatrix dst, float degrees, float x, float y,
    float z, pfMatrix m);
void   pfPreScaleMat(pfMatrix dst, float x, float y, float z,
    pfMatrix m);
```

Each of these methods performs the matrix math to concatenate one matrix transformation with another before the SCS node is traversed by an action.

For example, to rotate a shape and then translate it, use the following code:

```
pfMakeRotMat(mat, degrees, x, y, z);
pfPreTransMat(mat, x1, y1, z1, mat);
```

Optimizing SCS Transformations

When you have multiple SCS transformations in a branch of a hierarchy, you can optimize the performance of an application by pre-calculating their concatenation. To do so, use the following pfNode methods:

```
int pfFlatten(pfNode *node, int mode);
pfNode *pfdCleanTree(pfNode *node, pfuTravFuncType func);
```

The *mode* argument in pfFlatten is currently ignored and should be 0.

pfFlatten

pfFlatten precalculates at initialization the result of all transformation matrices in a branch of a node hierarchy. If a geometry is referenced by more than one SCS node, pfFlatten does the following:

1. Clones the geometry for each SCS node.
2. Calculates the transformed locations of each SCS node.
3. Changes the SCS matrix values to an identity matrix.

Figure 9-4 shows this process.

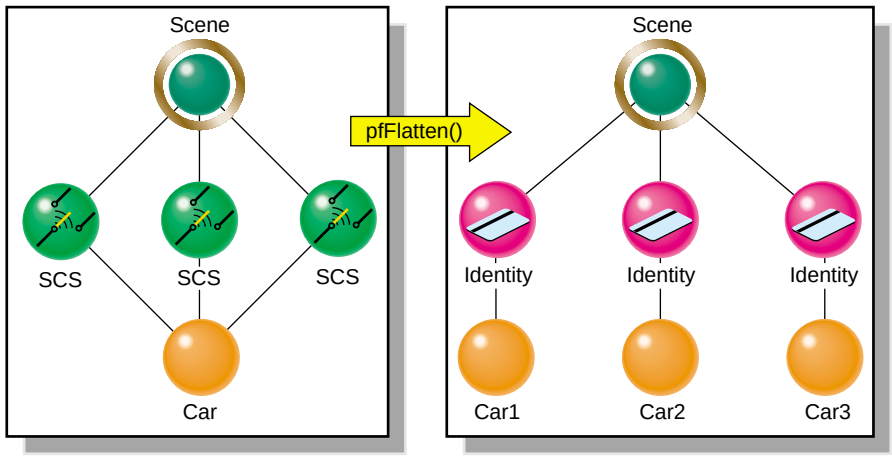

Figure 9-4 pfFlatten

In Figure 9-4, pfFlatten calculates the transformation of the car into three locations; those locations are stored in the Scene node. As a result, three matrix calculations are reduced to matrix result.

You identify the node in which to store the matrix concatenation in the pfFlatten method. This node is generally the node at the top of a branch.

Note: If a DCS node is encountered under the SCS node, an SCS node is inserted above the DCS node.

Flattening can substantially improve performance, especially when many pfSCS nodes are parents for a relatively small number of geometries. However, as Figure 9-4 shows, pfFlatten can also increase the size of the database. To remedy that problem, you use pfdCleanTree, as described in “pfdCleanTree” on page 153.

Flattening also increases the ability of OpenGL Performer to sort the database by mode, often a major performance enhancement, because sorting does not cross transformation boundaries.

pfdCleanTree

The SCS nodes containing identity matrices as a result of **pfFlatten()** serve no function. To remove these nodes from the database, as shown in Figure 9-5, use pfdCleanTree using NULL as the value of *func*.

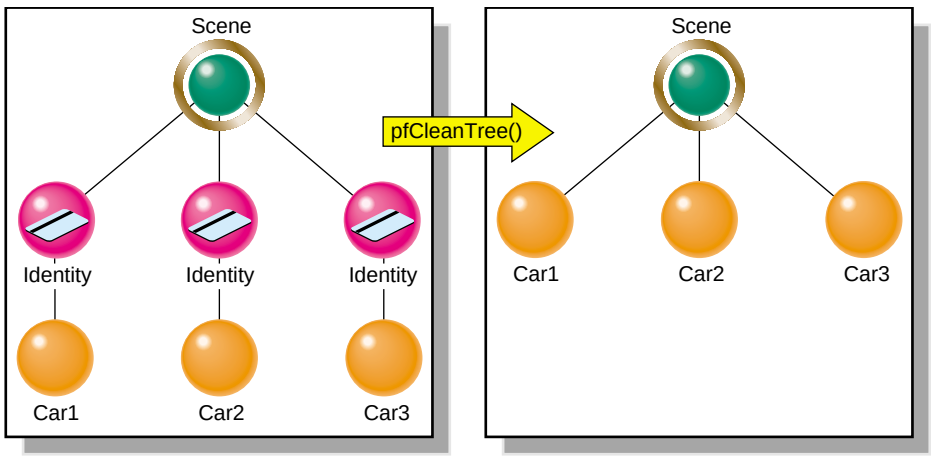

Figure 9-5 pfdCleanTree

When *func* is NULL, pfdCleanTree performs as follows:

1. Converts pfSCS nodes with identity matrices into pfGroup nodes.
2. Removes any pfGroup nodes with zero or one child.

   One exception is a pfSwitch node with one child, which is not eliminated.

You only call pfdCleanTree after calling pfFlatten.

Optionally, you can supply your own function to change the behavior of pfdCleanTree. Whatever the function, if *func* returns TRUE, the current node is eliminated; if *func* returns FALSE, the current node is retained.

Controlling Frame Rate

Frame rate is the number of times a scene is redrawn per second. Frame rate is constrained by three factors:

- Rate at which the screen is refreshed.
- Specified frame rate.
- Time required to calculate and draw the scene.

For example, one system may have a refresh rate of 60 frames per second. Other systems may have frame rates limited to the frame rate divided by an integer, for example, 30, 20, 15, 12, and 10 frames per second.

This chapter describes how to control the frame rate in the following sections:

Double Buffering

OpenGL Performer uses the standard double buffering mechanism for displaying scenes:

- The front buffer sends a complete description of the scene to the graphics pipeline.
- The back buffer is filled with the next frame of information to be displayed.

At a frame boundary, if the drawing to the back buffer is complete and a swap buffer has been issued, the front and back buffers are swapped so that:

- The back buffer becomes the new front buffer whose graphic information is scanned out.
- The front buffer becomes the new back buffer to hold the next frame, as shown in Figure 10-1.

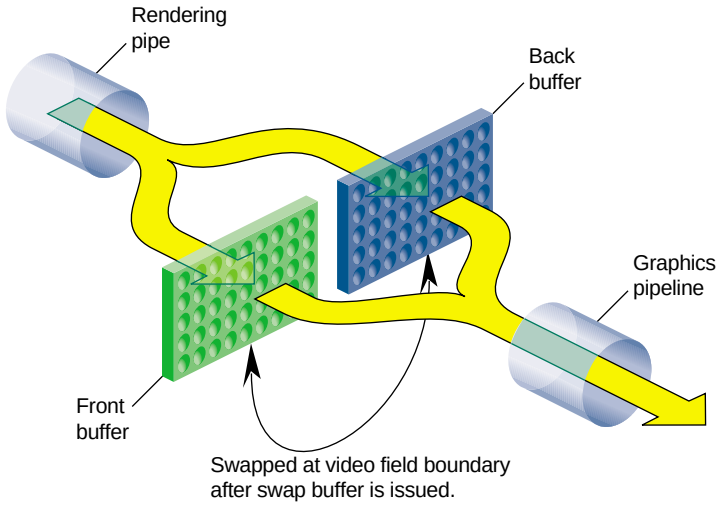

Figure 10-1 Double Buffering

Specifying a Target Frame Rate

You can only specify a target frame rate, not the frame rate, because sometimes calculating and drawing a frame can require more time than the time between screen refreshes. If a screen is not entirely drawn, rather than drawing part of a scene, the current frame is redisplayed while the drawing of the next scene completes.

You can specify the target frame rate using one of two methods:

- **pfFrameRate()**
- **pfFieldRate()**

pfFrameRate

You can set the target frame rate directly using the following pfFrame method:

```
void pfFrameRate(float rate);
```

rate is rounded to the nearest frame rate that corresponds to an integral number of screen refreshes, for example, a value of 33 frames per second (FPS) is rounded to 30 FPS.

The target time required to draw a frame is the reciprocal of the frame rate.

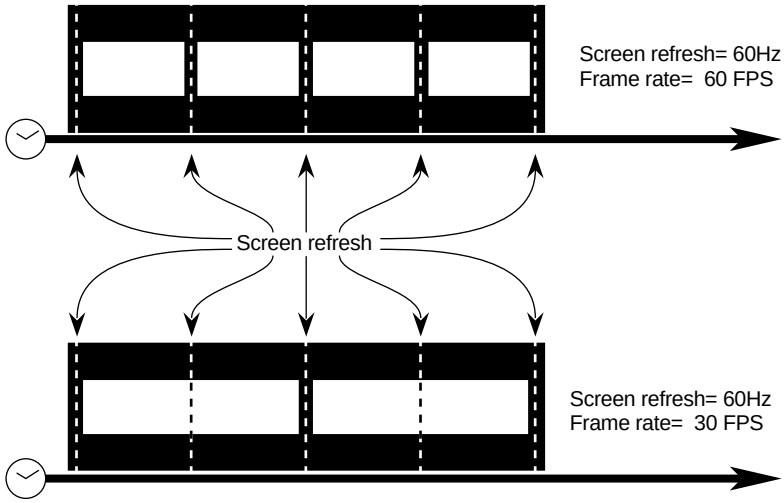

Figure 10-2 Frame Rate

With a screen refresh rate of 60 Hz, Figure 10-2 shows the frame boundaries for two different frame rates.

pfFieldRate

You can set the frame rate indirectly using the following pfFrame method:

```
void pfFieldRate(int fields);
```

fields refers to the number of screen refreshes per frame. The corresponding frame rate is the video field rate divided by *fields*.

Frame Synchronization

pfSync synchronizes the graphics pipeline to the frame rate. This method makes all processes start on frame boundaries. This keeps computations between multiple processes consistent and based on data, such as eyepoint, rather that computation time. Because computation time is variable, basing motion on the completion of computations thereby creates unsmooth motion. Moving at the start of frame boundaries produces smoother motion.

Exactly how pfSync responds to DRAW time overruns is specified by the phase control.

pfFrame, which sets off processes (APP, CULL), calls pfSync automatically if the user has not called it for the current frame.

Phase Control

When drawing of the scene is complete in the background buffer, use pfPhase to specify when to display the next frame (when it takes longer than the refresh rate to draw a scene.)

PHPHASE_FREE_RUN
: tells the application to run as fast as possible—to display each new frame as soon as it is ready, without attempting to maintain a constant frame rate.

PFPHASE_LIMIT
: tells the application to run as fast as possible, but the rendering rate is limited to the frame rate specified by pfFrameRate.

PHPHASE_FLOAT
: allows the drawing process of a new frame (using **swapbuffers**(3G)) to begin at any time, regardless of frame boundaries, but the display of the frame is synchronized with the next frame boundary. If the DRAW extends beyond the frame boundary, APP can continue. Application frames might get skipped by DRAW, which is asynchronous.

PHPHASE_LOCK
: requires the DRAW process to wait for a frame boundary before displaying a new frame.

Figure 10-3 shows these four options.

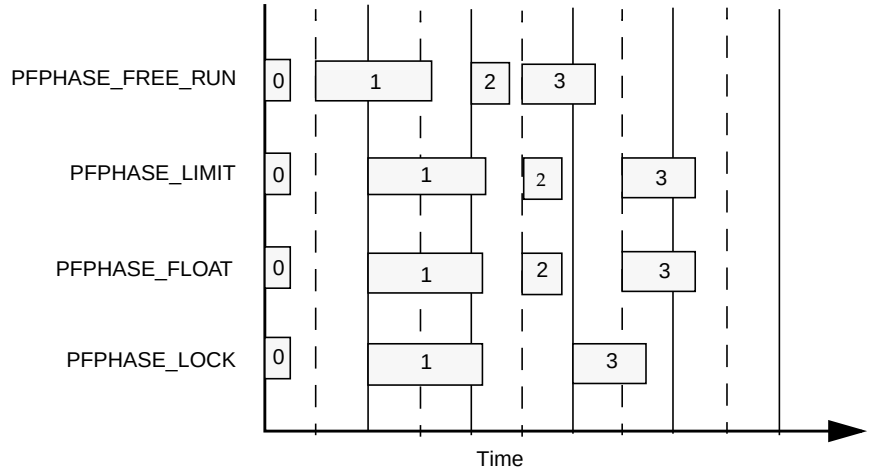

Figure 10-3 Phase Control over Three Frames

Note the following in Figure 10-3:

- Screen refresh = 60 Hz.
- Frame rate = 30 Hz.
- Frame 1 requires too much time to draw.

Adjusting the Frame Rate Automatically

Erratic frame rates cause jumpy images. Rather than changing frame rates according to whether or not a scene is drawn quickly enough, OpenGL Performer uses two mechanisms to smooth out frame rates:

- Stress filters in pfChannel.
- Dynamic Video Resolution (DVR).

Stress Filters

pfChanStressFilter() automatically adjusts the level of detail (LOD) displayed according to the speed at which the frames are being processed. As long as the speed is within a range of values, the stress level and the LODs displayed remain the same. If the stress level falls below that range, the LOD is increased. If the stress level moves above the accepted stress range, the LOD is decreased.

pfChanStress() allows you to handle the LOD display manually.

For more information about stress filters, see Chapter 5, "Frame and Load Control," in the *OpenGL Performer Programmer's Guide*.

Dynamic Video Resolution

On InfiniteReality machines, you can use Dynamic Video Resolution (DVR) to help maintain a constant frame rate. The methods in pfPipeVideoChannel monitor how much time is required to draw each frame. If the frame takes too long to draw, the size of the pfChannel is reduced so that it requires fewer pixels to render. The output is then scaled back to the correct size, so the image appears to be the correct size. If the frame requires too little time to draw; the video output is not reduced.

When using DVR, the origin and size of a channel are dynamic. For example, a viewport whose lower-left corner is at the center of a pfPipe (with coordinates 0.5, 0.5) would be changed to an origin of (0.25, 0.25) with respect to the full pfPipe window if the DVR settings were scaled by factors of 0.5 in both X and Y dimensions. This allows fewer pixels to be drawn per pfChannel for a faster rendering of the scene. Video hardware automatically rescales the image to full size with no penalty or added latency.

Setting the DVR Stress Filter

To set the stress filter, use **psPVChanStressFilter,** defined as follows:

```
void
pfPipeVideoChannel::psPVChanStressFilter(pfPipeVideoChannel*pv,float
*frameFrac,float *lowLoad, float *highLoad, float *pipeLoadScale,
float *stressScale, float *maxStress);
```

frameFrac is the fraction of a frame that pfPipeVideoChannel is expected to require to render the frame. For example, if the rendering time is equal to the period of the frame rate, *frameFrac* is 1.

If there is only one pfPipeVideoChannel, it is best to set *frameFrac* to 1. If there is more than one pfPipeVideoChannel, it is best to divide *frameFrac* among the pfPipeVideoChannels, such that a channel rendering complex scenes is allocated more time than a channel rendering simple scenes.

For more information about DVR, see Chapter 5, “Frame and Load Control,” in the *OpenGL Performer Programmer’s Guide*.

Multiprocessing

You can achieve higher frame rates by processing image data on multi-CPU platforms. Each stage of the graphics pipeline process can then run as a separate process on a separate CPU. Each pipeline can handle up to five processes. Although you can construct the processes as you like, the recommended processes include three synchronous processes:

- APP—for updating node values.
- CULL—for eliminating from rendering calculations any nodes outside of the view frustum.
- DRAW—for rendering shapes.

The three recommended asynchronous processes include:

- ISECT—for intersection calculations.
- DBASE—for paging image data into system memory.
- COMPUTE—for general, asynchronous computations.

This chapter describes how to use multiprocessing in the following sections:

OpenGL Performer Stages

The APP, CULL, and DRAW stages comprise the required stages of the graphic pipeline. There can be only one APP process for an application. There are, however, separate pairs of CULL and DRAW stages for each pfPipe, as shown in Figure 11-1.

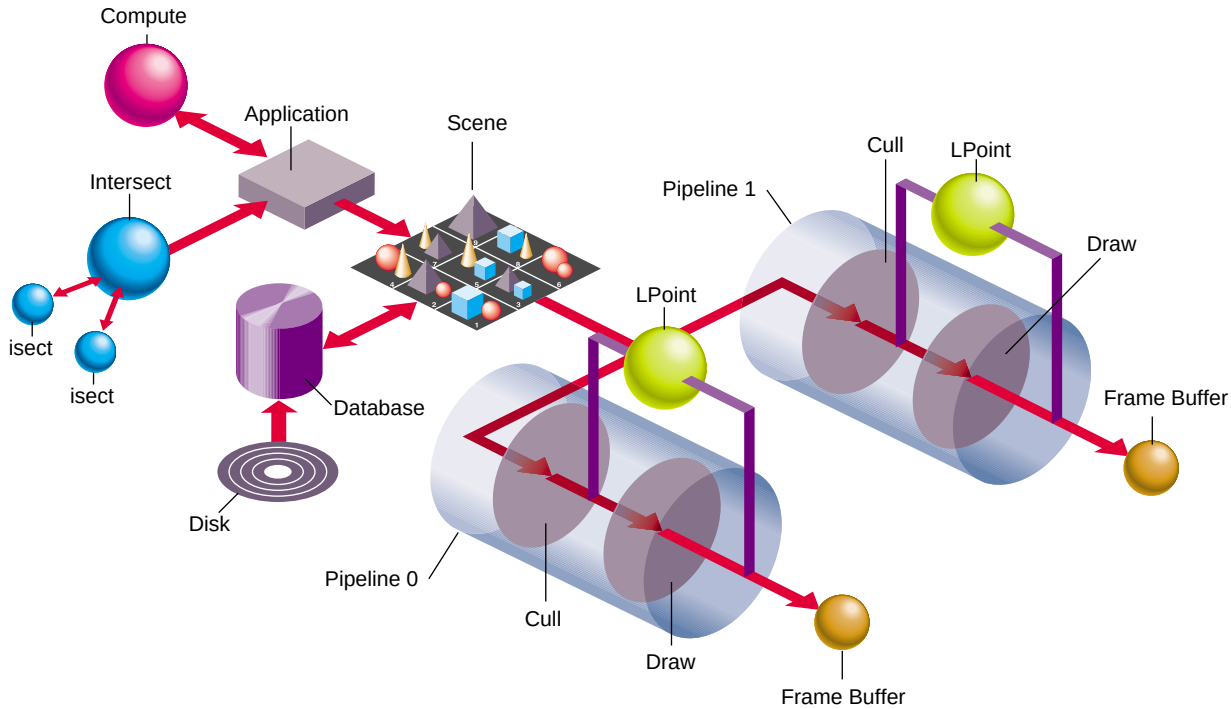

Figure 11-1 OpenGL Performer Stages

You can change the default behavior of the DRAW and CULL stages using callback functions.

Optional, Asynchronous Stages

If you do not fork off separate processes for intersection testing (ISECT), I/O (DBASE), or miscellaneous calculations (COMPUTE), the calculations are performed in the APP stage and will be performed serially.

Each of the asynchronous stages perform computationally intense calculations in parallel with the required stages to improve the overall speed of image processing.

ISECT Stage

The ISECT stage calculates intersection-related information. To do that calculation, it keeps a copy of the scene graph. Consequently, this stage can use a significant amount of memory, depending on the size of the scene graph.

For more information about intersection testing, see Chapter 13, “Intersection Testing.”

DBASE Stage

The DBASE stage deals with I/O issues of downloading scene graph data from the hard drive to system memory. This stage is lightweight because it does not keep a copy of the scene graph.

For more information about the DBASE process, see Chapter 12, “Database Paging.”

COMPUTE Stage

The COMPUTE stage is provided for general calculations. It does not contain a copy of the database, but it does contain general statistics and the number of the frame that is being processed.

When you fork off this process, pfASD is computed in this stage as is pfFlux, in addition to any calculations you place in this stage.

Benefits of Multiprocessing

Multiprocessing enables parallel processing of image data in the graphics pipeline. If each of the three stages in the graphics pipeline, (APP, CULL, and DRAW) run sequentially, and each take 16 milliseconds, each frame would require 48 milliseconds for processing. If, however, each stage is processed in parallel, the processing time for a single frame is reduced to 16 milliseconds, as shown in Figure 11-2.

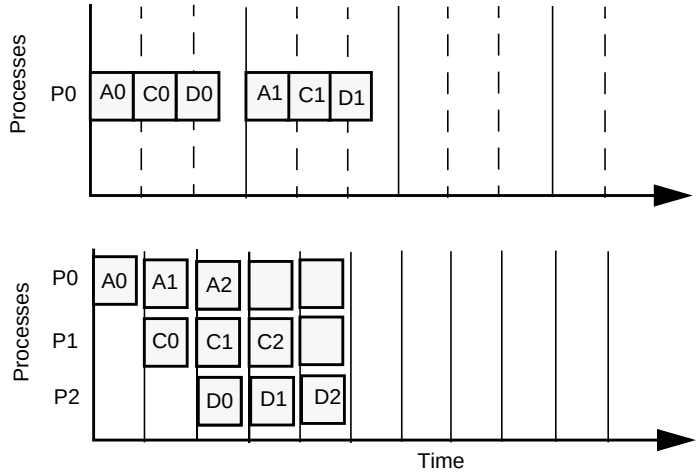

Figure 11-2 Multiprocessing in the Graphics Pipeline

Figure 11-2 shows that three tasks running sequentially (in the upper figure) require three times the processing time of the three tasks running in parallel (in the lower figure), each in their own process.

The shorter processing time dramatically affects the frame rate at which the application can display its images.

Shared Memory

The shared memory arena contains a copy of the frame's data that is used by each process, in the following way:

1. After the APP process updates the frame, the process places a copy of unique data for the frame in the shared memory arena.
2. The CULL process takes the frame from the shared memory arena, culls out data invisible to the viewer, and places a revised copy of the frame back in the shared arena memory in the form of a `libpr` display list for that frame.
3. The DRAW process uses the updated frame and renders the scene to the display system.

Figure 11-3 shows how the shared memory arena is used by the different stages.

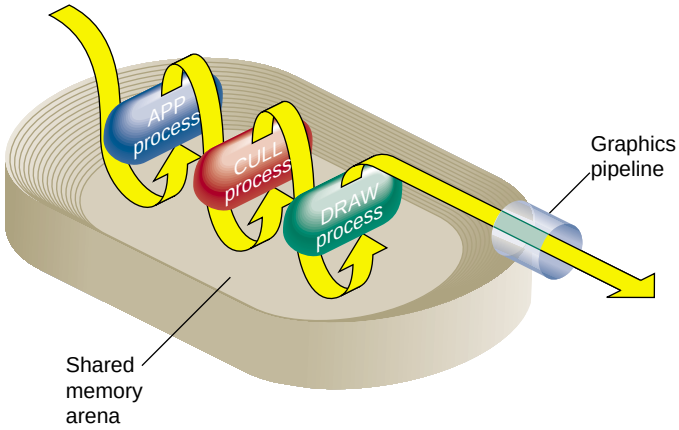

Figure 11-3 Shared Memory Arena

Printing Process States

pfPrintProcessState() prints a description of OpenGL Performer processes to a file. The following shows a sample printout:

```
Proc: APP       pid:11895
Proc: ISECT     pid:11895
Proc: DBASE     pid:11895
Proc: CLOCK     pid:11896
Proc: COMPUTE   pid:11895
Proc: SYNC      pid:0
Pipe Proc: CULL Pipes:1
        Thread Proc: CULL Pipe:0        Threads:0
        Parent:Proc: CULL Pipe:0        pid:0
Pipe Proc: DRAW Pipes:1
        Proc: DRAW Pipe:0       pid:0
Pipe Proc: LPOINT       Pipes:1
        Thread Proc: LPOINT Pipe:0      Threads:0
        Parent:Proc: LPOINT Pipe:0      pid:0
```

Setting Up Multiprocessing

OpenGL Performer simplifies setting up multiple processes by supplying the tokens shown in Table 11-1 for the following pfConfig method:

```
int pfMultiprocess(int mode);
```

mode is one or more multiprocessing models ORed together. Table 11-1 lists the tokens to use for *mode*. These processing models are set at creation time and cannot be altered at run time.

You call pfMultiprocess between pfInit and pfConfig.

Multiprocessing Models

Table 11-1 lists the multiprocessing models available in OpenGL Performer.

Table 11-1 Multiprocessing Tokens

Token	Description
PFMP_DEFAULT	Chooses a multiprocessing mode based on the number of pipelines required and the number of unrestricted, available processors.
PFMP_FORK_ISECT	Fork an asynchronous ISECT process.
PFMP_FORK_CULL	Place CULL in a separate process.
PFMP_FORK_DRAW	Place DRAW in a separate process.
PFMP_FORK_DBASE	Fork an asynchronous DBASE process.
FMP_FORK_COMPUTE	Fork an asynchronous COMPUTE process.
PFMP_CULLoDRAW	Overlap CULL and DRAW processes.
PFMP_CULL_DL_DRAW	Force CULL to generate display list.
PFMP_APPCULLDRAW	All stages are combined into a single process. A pfDispList is not used. pfDraw both culls and renders the scene.
PFMP_APPCULL_DL_DRAW	All stages are combined into a single process. A pfDispList is built by pfCull and rendered by pfDraw.

Table 11-1 Multiprocessing Tokens **(continued)**

Token	Description
PFMP_APP_CULLDRAW	The CULL and DRAW stages are combined in a process that is separate from the application process. A pfDispList is not used. pfDraw both culls and renders the scene. Equivalent to (PFMP_FORK_CULL).
PFMP_APP_CULL_DL_DRAW	The CULL and DRAW stages are combined in a process that is separate from the application process. A pfDispList is built by pfCull and rendered by pfDraw. Equivalent to (PFMP_FORK_CULL | PFMP_CULL_DL_DRAW).
PFMP_APPCULLoDRAW	The APP and CULL stages are combined in a process that is separate from, but overlaps, the DRAW process. Equivalent to (PFMP_FORK_DRAW | PFMP_CULLoDRAW).
PFMP_APP_CULL_DRAW	The APP, CULL, and DRAW stages are each separate processes. Equivalent to (PFMP_FORK_CULL | PFMP_FORK_DRAW).
PFMP_APP_CULLoDRAW	The APP, CULL, and DRAW stages are each separate processes and the CULL and DRAW process are overlapped. Equivalent to (PFMP_FORK_CULL | PFMP_FORK_DRAW | PFMP_CULLoDRAW).
PFMP_FORK_LPOINT	Fork a light process, pfLPointState.

The “o” in PFMP_CULLoDRAW stands for “overlap.” The CULL and DRAW processes can overlap when they are separate. Figure 11-4 shows that the DRAW process acts on the first frame one screen refresh earlier in the PFMP_CULLoDRAW model than in the PFMP_APP_CULL_DRAW model.

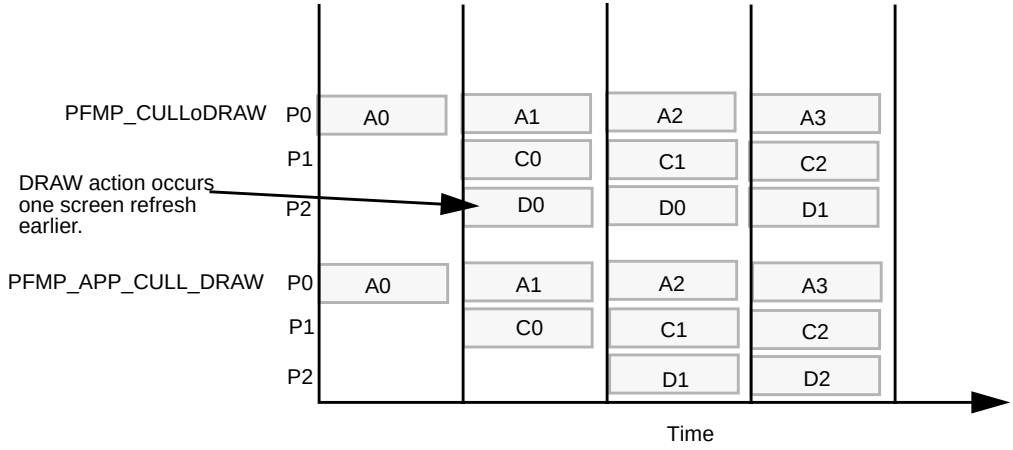

Figure 11-4 PFMP_CULLoDRAW

Common Multiprocessing Models

Figure 11-5 shows four common multiprocessing models.

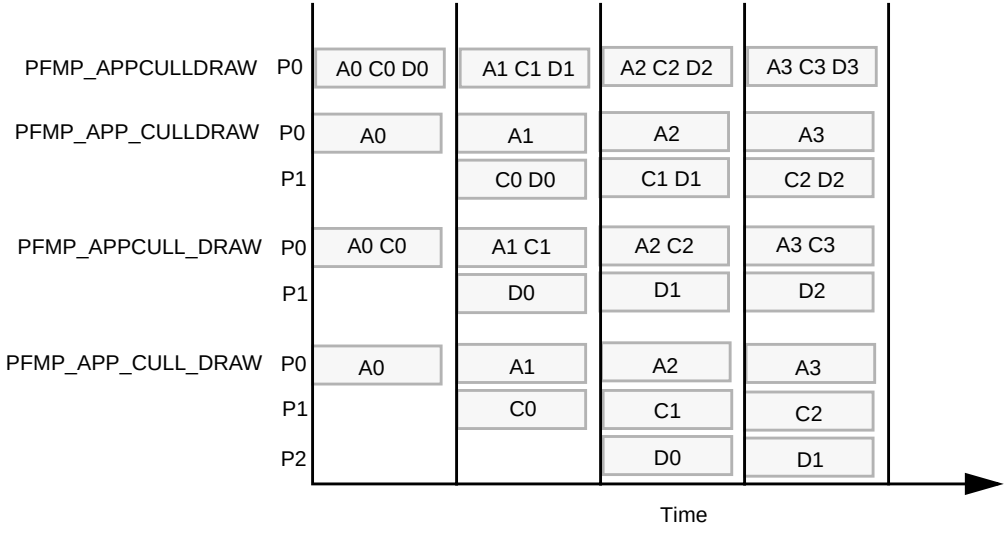

Figure 11-5 Four Common Multiprocessing Models

Tip: In two-processor mode, fork off the stage that consumes the most time.

Default Processing Models

The default multiprocessing model set up by PFMP_DEFAULT depends on the following:

- Number of pfPipes
- Number of unrestricted CPUs

One pfPipe

If there is one pfPipe in the system, the default multiprocessing model depends upon the number of unrestricted CPUs, as described in Table 11-2.

Table 11-2 Default Multiprocessing Models

Number of CPUs	Default Multiprocessing Model
1	PFMP_APPCULLDRAW
2	PFMP_APPCULL_DRAW
3	PFMP_APP_CULL_DRAW

Multiple pfPipes

When multiple pfPipes are configured, the default multiprocessing model always defaults to pfPipe::PFMP_APP_CULL_DRAW. In multiprocessing models, the CULL process must be separate from the APP process.

Choosing a Multiprocessing Model

An application only runs as fast as its slowest stage. To improve the performance of your application, you need to determine which stage acts as a bottleneck. Generally, of the three synchronous processes, the DRAW stage takes the most time. Place the stage that requires the longest time in its own process.

Automatic Multiprocessing

When you enable the Process Manager, pfuProcessManager, found in `libpfutil`, it automatically evaluates the number of processes and processors that you have and spreads the processes evenly over the processors. You enable pfuProcessManager with the routine **pfuInitDefaultProcessManager()**.

Note: pfuProcessManager obsoletes pfuLockCPU.

Database Paging

Many scene graphs you use are too large to fit into system memory. Consequently, you need to load data dynamically at run time. Because loading data from a hard drive is relatively slow, to prevent breaking the frame rate, you should:

- Fork off a database (DBASE) process to handle database paging asynchronously in the background.
- Anticipate which pages of data to load and which to delete.

This chapter describes how to page the database efficiently in the following sections:

Anticipating Paging

Because many scene graphs are too large to hold into system memory, your application must anticipate which pages of memory to load and which to delete. Pages of memory are often associated with nodes in the scene graph: a node encapsulates a part of the scene which occupies a page of memory.

Figure 12-1 shows pages of memory represented as squares; each page corresponds to a node in the scene graph. The triangle represents the position and direction of the motion of the eyepoint.

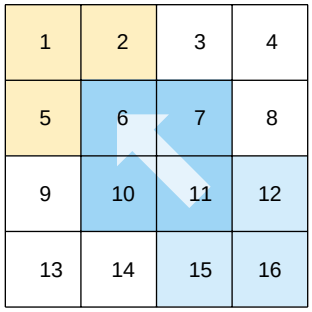

Figure 12-1 Memory Pages

In Figure 12-1, pages 6, 7, 10, 11, 12, 15, and 16 are currently in memory, pages 1, 2, and 5 are good candidates for loading, and pages 12, 15, and 16 are good candidates for removal.

Database Process

Because loading data from disk is relatively slow, loading deserves its own process so that it can run continuously and asynchronously in the background.

To use a database paging process, you must:

- Fork off a DBASE process.
- Call a database function of your creation, which handles memory allocation and deallocation, and the loading of the data.

The following code performs those tasks:

```
pfInit();
pfMultiprocess( PFMP_FORK_DRAW | PFMP_FORK_DBASE );
pfConfig();

pfDBaseFunc( myDBaseFunc );
```

myDBaseFunc, of type `pfDBaseFuncType`, needs to handle data loading and memory allocation and deallocation.

Handling Memory for the DBASE Process

The APP and DBASE processes need to share data. They reside, however, in separate virtual memory spaces. To share data, they must allocate memory in the shared memory arena, as the following code shows:

```
typedef struct (
    pfScene *Scene;
) SharedData;

SharedData *shared;
void *arena;

arena = pfGetsharedArena();
shared = (SharedData *)pfMalloc( sizeof(SharedData), arena );

shared->scene = pfNewScene();
```

These lines of code, except for the last, must be placed between pfInit and pfConfig. To deallocate the memory malloc'd, use pfFree.

The final line of code makes the scene node, the root node of the scene graph, accessible to the DBASE process.

Changing the Scene Graph

Because of user interaction, such as moving through a scene, the scene graph in memory often changes: nodes representing pages of memory are deleted or added to the scene graph according to the motion of the eyepoint. The DBASE process should not change the scene directly because it should anticipate where the eyepoint will go. If the process were to change the scene graph immediately, the anticipated page of memory would likely display too soon. Instead, the DBASE process should:

1. Cache scene graph changes in a pfBuffer.
2. Add and remove nodes from the scene graph in the buffer.
3. Delete nodes removed from the scene graph in the buffer.
4. Merge the changes from the buffer into the scene graph when the APP process calls pfSync.
5. Carry out the deletion request.

The following sections explain how to perform these tasks.

Caching Scene Graph Changes

Instead of changing the scene graph directly, you should:

1. Create a buffer.
2. Make it active.
3. Create the necessary scene graph changes in the buffer.

The following lines of code complete these tasks.

```
pfBuffer *buf;
node *d, *e;

buf = pfNewBuffer();
pfSelectBuffer( buf );
d = pfNewGroup();
e = pfNewGeode();
pfAddChild( d, e );
```

Figure 12-2 shows how a buffer is created and how nodes are created and grouped.

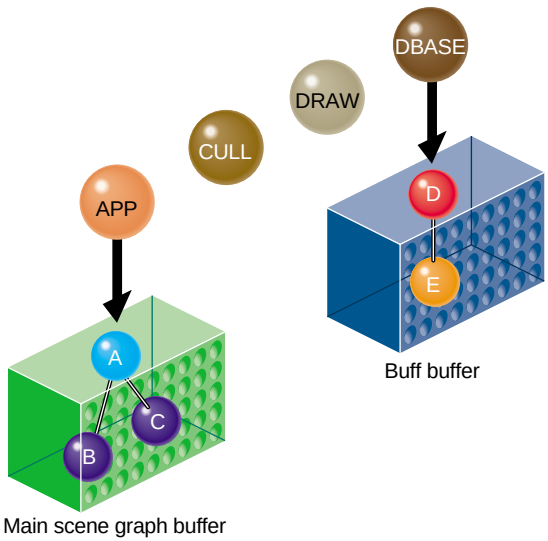

Figure 12-2 Creating the Buffer and Changes

Linking Buffer Changes to the Scene Graph

Once the scene graph in the buffer is complete, you must connect the changes to the main scene graph. To remove node C and connect the scene graph in the buffer to node A, use the following lines of code:

```
pfBufferRemoveChild( a, c );
pfBufferAddChild( a, d );
```

These lines of code request but do not cause the actions to be performed. The actions are performed with the next call to pfSync.

Deleting Old Data

Once you request a node to be removed, you should request that it be deleted as well. The following line of code removes node C:

```
pfAsyncDelete( c );
```

This code requests the deletion, but the deletion is not performed until the next call to pfSync.

Figure 12-3 shows the linking and deletion of nodes.

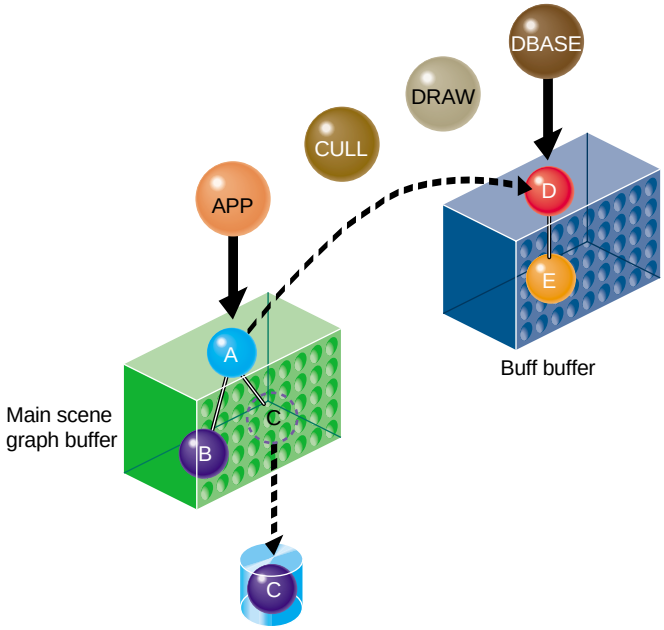

Figure 12-3 Linking and Deleting Nodes

Figure 12-3 shows that although node C was removed and deleted, its data remains in the cache.

Merging Changes

To make the changes to the main scene graph take effect when **pfSync()** is called, you must merge the changes, as follows:

```
int success;

success = pfMergeBuffer();
```

success is non-zero if the merge is successful.

When you merge the buffers, the following occurs:

- Nodes D and E are placed in the scope of the main scene graph buffer.
- The buff buffer is cleared.

Figure 12-4 shows these changes.

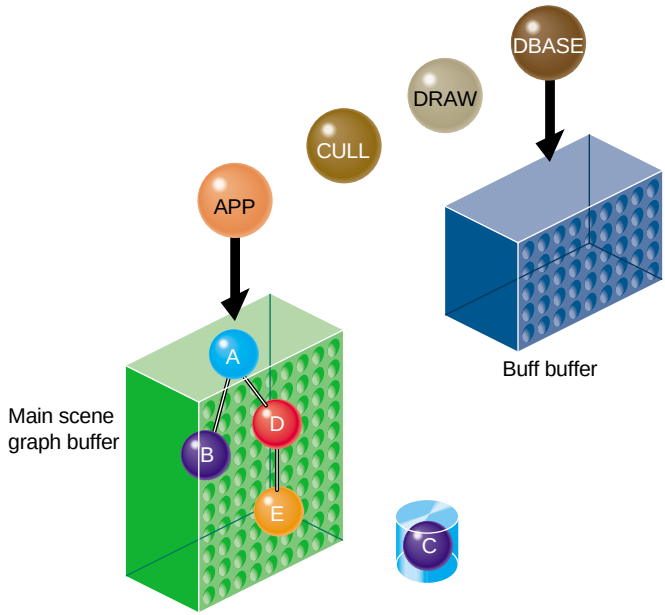

Figure 12-4 Merging Scene Graph Changes

The merge is not performed until the next call to pfSync.

Cleaning Up the Cache

To completely delete removed nodes from system memory, but not the hard drive, call the following method from the DBASE process:

```
void pfDBase(void);
```

This method carries out the deletion specified in **pfAsyncDelete()**. Be sure to call it after **pfMergeBuffer()**.

Intersection Testing

Detecting when one shape touches another is useful for determining:

- Collision detection
- Terrain following
- Shape selection
- Orientation according to terrain

Instead of using a bounding volume for intersection testing, you construct segments that approximate the shape of the object, as shown in Figure 13-1.

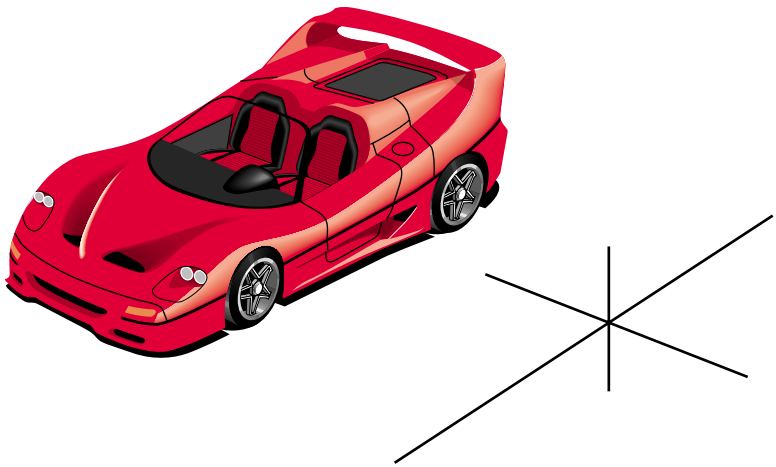

Figure 13-1 Approximating a Shape with Segments

This chapter describes how to check for intersections in the following sections:

For more information on intersections, see Chapter 4, "Database Traversal," in the *OpenGL Performer Programmer's Guide*. Also, there are several source code examples included in the OpenGL Performer source code for intersections:

- `/usr/share/Performer/src/pguide/libpf/C/intersect.c`
  (IRIX and Linux)
  `%PFROOT%\Src\pguide\libpf\C\intersect.c`
  (Windows)
  This example demonstrates basic intersection functionality.
- `/usr/share/Performer/src/lib/libpfutil/collide.c`
  (IRIX and Linux)
  `%PFROOT%\Src\lib\libpfutil\collide.c`
  (Windows)
  This example uses intersections to implement ground-following and object collision for a moving vehicle.
- `/usr/share/Performer/src/lib/libpfui/pfiPick.C`
  `/usr/share/Performer/src/pguide/libpfui/pick.c`
  (IRIX and Linux)
  `%PFROOT%\Src\lib\libpfui\pfiPick.C`
  `%PFROOT%\Src\pguide\libpfui\pick.c`
  (Windows)
  These use **pfChanPick()**, which is based on intersections.

Creating an ISECT Process

By default, no intersection processing is performed by OpenGL Performer. OpenGL Performer allows you to specify a function that will be called in the intersection stage of the application with **pfIsectFunc()**.

```
void pfIsectFunc(pfIsectFuncType func);
```

Intersection testing is often time consuming. For that reason, it is often a good idea to let the task run asynchronously in its own process. If the PFMP_FORK_ISECT bit is specified in the **pfMultiprocess()** bitmask, the intersection stage will be run as a separate asynchronous process and therefore can extend beyond the time for a frame without impacting the performance of the main rendering pipeline.

```
pfInit();
pfMultiprocess(PFMP_FORK_DRAW | PFMP_FORK_ISECT | PFMP_FORK_DBASE);
pfConfig();

pfIsectFunc(myIsectFunc);
```

Because the ISECT process runs asynchronously, two objects could collide without providing immediate notification of the collision. If immediate notification is very important, perform those intersections separately as part of the application process, or do not create a separate intersection process. The function you register with pfIsectFunc is called from the APP process and runs synchronously.

Constructing a Segment Set for pfNodeIsectSegs()

Evaluating every point on the surface of a geometric surface is far too computation-intensive. Instead, a set of segments is used to grossly approximate the shape of an object to test for intersections with that object. **pfIsectNodeSegs()** traverses a scene graph looking for intersections with the provided segment set.

To create a segment set, you create a pfSegSet, which is a structure defined as follows:

```
typedef struct {
    int mode;
    void *userData;
    pfSeg segs[PFIS_MAX_SEGS]; /* currently 32 */
    uint activeMask;
    uint isectMask;
    void *bound;
    int (*discFunc)(pfHit *);
} pfSegSet;

typedef struct {
    pfVec3 pos;
    pfVec3 dir;
    float length;
} pfSeg;
```

Setting the Mode

The mode field of pfSegSet specifies the kind of information recorded when there is a hit at an intersection. The mode value is a bitwise OR of one or more of the values in Table 13-1.

Table 13-1 Segment Set Modes

Mode	Description
PFTRAV_IS_PRIM	Intersect with quads or triangle geometry.
PFTRAV_IS_GSET	Intersect with pfGeoSet or pfGeoArray bounding boxes.
PFTRAV_IS_GEODE	Intersect with pfGeode bounding sphere.
PFTRAV_IS_NORM	Return normals in the pfHit structure.
PFTRAV_IS_CULL_BACK	Ignore back-facing polygons.
PFTRAV_IS_CULL_FRONT	Ignore front-facing polygons.
PFTRAV_IS_PATH	Retain traversal path information.
PFTRAV_IS_NO_PART	Do not use partitions for intersections.

PFTRAV_IS_PRIM, PFTRAV_IS_GSET, and PFTRAV_IS_GEODE define where the geometry is stored and thereby define the depth of the intersection test.

Intersection Masks

There are several mask fields in the pfSegSet structure to enable you to have conditional intersection traversal. The activeMask field of pfSegSet is a 32-bit mask that allows you to specify which segments are active.

The isectMask is masked with the intersection mask of each node, set on the node by **pfNodeTravMask()** for PFTRAV_ISECT, in the intersection traversal. The traversal will not intersect with a node or its children if the AND of its isectMask and the pfSegSet isect Mask is zero. This allows you to create intersection classes of objects and intersection types.

pfNodeTravMask() allows you to specify additional things for controlling the intersection traversal of a scene. The function prototype is:

```
pfNodeTravMask(pfNode *node, int which, uint mask, int setMode, int bitOp);
```

The setMode enables you to specify intersection caching, and the bitOp enables you to and- or- or set- the specified mask against the previous mask in the node. The following is an example call:

```
pfNodeTravMask(node, PFTRAV_ISECT, 0xffffffff,
PFTRAV_SELF|PFTRAV_DESCEND|PFTRAV_IS_CACHE /* turn on caching for
entire scene graph below object */,
PF_OR /* or- with prev bitmask in each node for new bitmask */);
```

Creating the Segment Array

The segment array defines the origin, direction, and length of the segments in the following structure:

```
typedef struct {
    pfVec3 pos;
    pfVec3 dir;
    float length;
} pfSeg;
```

The pfSegSet array can have up to 32 segments. The segment array is then set in the pfSegSet structure in the **segs[]** field.

The pfSegSet Bound

To further improve intersection performance when you have many segments in a pfSegSet, you can provide a bounding pfCylinder for the segments in the bound field of the pfSegSet. You can create the bounding cylinder can be created with **pfCylAroundSegs().** The array passed to **pfCylAroundSegs()** needs to be an array of pointers to pfSegs, which is different than the array of pfSegs for the pfSegSet.

Testing for Intersections

After setting up the segment set for the geometry and setting the mask for the nodes in the scene graph, you can check for intersections between the geometry and the segment set using the following pfNode method:

```
int pfNodeIsectSegs(pfNode *node, pfSegSet *segSet, pfHit **hits[]);
```

segSet is the segment set for the geometry you are testing.

node is where the intersection testing begins.

hits is an empty array that the traversal will fill with pfHit structures for successful intersections— one per segment.

Note: An alternative to **pfNodeIsectSegs()** is **pfChanNodeIsectSegs()**.

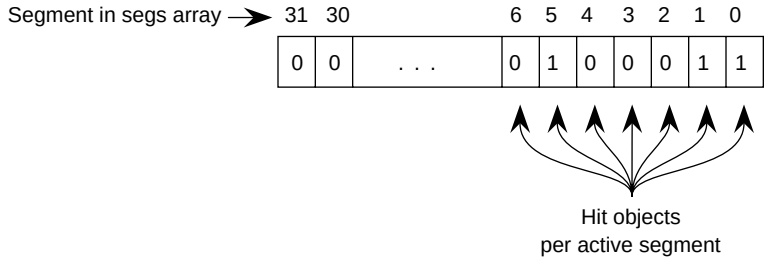

Figure 13-2 Hits Array

The number of array elements matches the size of the segment set array.

Intersection Information

The hits array is filled by the intersection traversal, with temporary hit structures for each intersection. The first element corresponds to the segment in the segment array, and the second dimension is a list of pointers to hit structures relating to that segment. The data in the hit structures only remains until another ISECT traversal is called, so you do not want to save pointers to the pfHit structures and you do not want to free them.

To return all of the other information contained in a Hit structure, use the following method:

```
int pfQueryHit(pfHit *hit, uint which, void *dst);
```

hit is a pfHit structure that the traversal fills with information regarding an intersection.

which is the information to retrieve from the hit structure.

dst is the pfMemory where the query results are placed.

Table 13-2 shows the PFHIT_ tokens, supplied for *which*, that specify the hit structure information to return.

Table 13-2 Hit Information

Token	Definition
PFQHIT_FLAGS	Indicates the validity of information in the hit structure described in this table.
PFQHIT_POINT	Returns the point of intersection.
PFQHIT_NORM	Returns the normal of the triangle at that point.
PFQHIT_SEG	Returns the current segment as clipped by the intersection process.
PFQHIT_PRIM	Provides the index of the primitive within the pfGeoSet or pfGeoArray.
PFQHIT_TRI	Returns the triangle index within the primitive.
PFQHIT_VERTS	Returns the vertices of the intersected triangle.
PFHIT_XFORM	Returns the non-identity transformation matrix.
PFQHIT_GSET	Returns a pointer to the pfGeoSet or pfGeoArray.
PFQHIT_NODE	Returns a pointer to the parent pfGeode.
PFQHIT_PATH	Returns a pfPath* denoting the traversal path.

PFQHIT_FLAGS is formed by optionally bitwise ORing zero or more of the PFHIT_POINT, PFHIT_NORM, PFHIT_PRIM, PFHIT_TRI, PFHIT_VERTS and PFHIT_XFORM symbols.

Determining If a Segment Was Hit

pfNodeIsectSegs() returns the number of segments that were intersected. To determine which of the segments were intersected, use code similar to the following:

```
uint flags;
pfHit **hits[32];

// Assume segset has four active segs, indicies 0 - 3

nhits = pfNodeIsectSegs(scene, &segset, hits);
```

```
for (i = 0, i < 4 && nhits > 0, i++){
    pfQueryHit(hits[i][0], PRQHIT_FLAGS, &flags);
    if (flags & PFHIT_ISECT) {
        // valid intersection
        nhits--;
    }
}
```

Testing for Valid Information

Each element in the hit array contains all of the hit structures recorded for a specific segment. All of the fields in the hit structures may not have data. To see if the fields have data, use the PFQHIT_FLAGS token in **pfQueryHit()**. Use this function to test the following fields:

- PFHIT_POINT
- PFHIT_NORM
- PFHIT_PRIM
- PFHIT_TRI
- PFHIT_VERTS
- PFHIT_XFORM

Use code similar to the following to test whether or not the fields have values; if not, they contain NULL:

```
pfQueryHit(hits[i][0], PFQHIT_POINT, &pt);
pfQueryHit(hit, PFQHIT_FORM, xform)'
if ( (flags & PFHIT_XFORM) == 0) {
    // xform contains garbage. so set it to the identity
    // matrix so there is no transformation.
    pfMakeIdentMat(xform);
}
pfXformPt3(xpt, pt, xform);
```

Retrieving the Intersection Location

You can find out what object in the scene graph was hit, the location of the intersection on that object, the object normal at the point of intersection, a traversal path to that object in the scene graph, and more. All of the geometric data is expressed in local object coordinates. To transform the data into world coordinates, use:

- pfXformPt3 to transform the point of intersection.
- pfXformVec3 to transform the normal.

Use these methods with the PFQHIT_XFORM matrix, as follows:

```
if (flags & PFHIT_ISECT) {
    pfVec3 pt, xpt;
    pfMatrix xform;

    pfQueryHit(hits[i][0], PFQHIT_POINT, &pt);
    pfQueryHit(hit, PFQHIT_XFORM, xform);
    pfXformPt3(xpt, pt, xform);

    shared->zpos = xpt[PF_Z] + 0.7;
}
```

In this example, the eyepoint is 0.7 units above the geometry.

Creating a User Interface

Real-time user interaction with complex databases is one of OpenGL Performer's strengths. This chapter describes how to create a user interface in the following sections:

Traveling through a Scene

Often you want to allow the user to travel through a scene using an input device, such as a mouse, as a guide for the motion. OpenGL Performer includes the transformer class, pfiTDFXformer, for manipulating the eyepoint.

OpenGL Performer provides three models for interpreting input device events:

- Trackball—when the input device is a trackball and the input is translated into 3D motion.
- Drive—where mouse events are translated into 2D motion.
- Fly—where mouse events are translated into 3D motion.

Note: pfiTDFXformer is a subclass of pfiXformer, which can be extended for custom motion models.

To use the transformer class to interpret mouse events as the means by which a user moves through a scene, use the following procedure:

1. "Creating a Transformer" on page 192.
2. "Initializing the Transformer" on page 193.
3. "Setting Up Transformer Input and Output" on page 194.

4. “Updating the Channel” on page 194.
5. “Scaling the Motion” on page 195.

The following sections explain this procedure.

Creating a Transformer

To create a transformer, you must:

1. Initialize the utility library, `libui`, using:

   ```
   void pfiInit(void);
   ```

2. Create the transformer in the shared memory arena, as follows:

   ```
   pfiTDFXformer *pfiNewTDFXformer(void *arena);
   ```

3. Check that the shared memory arena is not NULL, as follows:

   ```
   if (pfGetSharedArena() = NULL) {
       // use memory allocated from the local heap
   }
   ```

The shared memory arena is that portion of memory that is available to all OpenGL Performer processes. Three common processes in OpenGL Performer are:

- APP
- CULL
- DRAW

Each process handles only part of the rendering. When a process is finished operating on one frame, it returns the data to the shared memory arena so that another process can grab and process it, as shown in Figure 14-1.

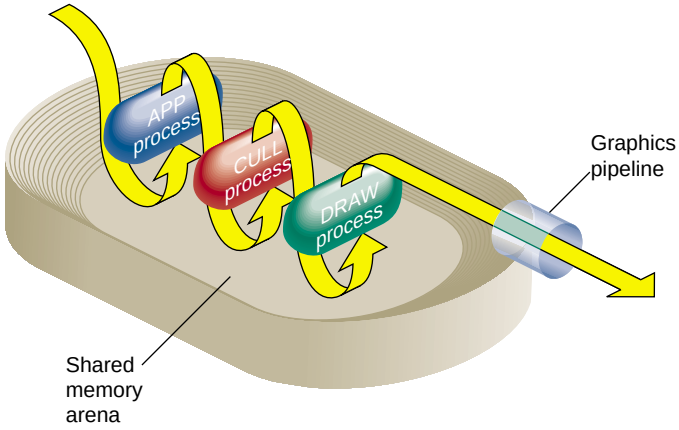

Figure 14-1 Shared Memory Arena

Initializing the Transformer

To initialize the transformer, you must:

1. Specify the motion model to use, using:

   ```
   void pfiSelectXformerModel(pfiXformer* XF, int model);
   ```

   where *XF* is the transformer created in the previous section, and *model* is one of three values:

   - PFITDF_TRACKBALL
   - PFITDF_DRIVE
   - PFITDF_FLY

   For more information on the motion models, see "Traveling through a Scene" on page 191.

2. Specify the initial position of the viewpoint, using:

   ```
   void pfiXformerCoord(pfiXformer* XF, pfCoord* coord);
   ```

   where *XF* is the transformer created in the previous section, and *coord* is a structure containing a three-dimensional set of coordinates, and three rotation values around those three dimensions. For more information about pfCoord, see "Direction and Position of the View" on page 83.

Setting Up Transformer Input and Output

To set up transformer input (using a mouse) and output, use the following steps:

1. Make the transformer object created previously, *XF*, read from the mouse and events buffer, as follows:

   ```
   void pfiXformerAutoInput(pfiXformer *XF, pfChannel *chan,
       pfuMouse *mouse, pfuEventStream *events);
   ```

   events points at the buffer containing the mouse events.

2. Specify the channel, *chan*, to update with the mouse events, as follows:

   ```
   void pfiXformerAutoPosition(pfiXformer* XF, pfChannel *chan,
       pfDCS *dcs);
   ```

   chan is updated for mouse events; *dcs* is updated for trackball events. If you do not want to interpret trackball input, set *dcs* to NULL.

3. Collect mouse events, as follows:

   ```
   pfuInitInput(mainPipeWindow, PFUINPUT_X);
   pfuCollectInput(void);
   pfuGetMouse(mouse);
   ```

   mainPipeWindow is the pfPipeWindow where the channel, *chan*, is rendered.

   PFUINPUT_X specifies that mouse and keyboard events are read from a forked process using X device commands. In this case, *mainPipeWindow*, must be a GLX window.

   mouse is a pointer to the buffer for mouse events.

Updating the Channel

To read the events from the input buffers and update the view in the channel, use the following method:

```
void pfiUpdateXformer(pfiXformer *XF);
```

where *XF* is the transformer.

This method updates the channel automatically.

Scaling the Motion

It is necessary to scale the effect of the motion of the mouse according to the size of the scene. For example, if moving the mouse an inch moves the viewpoint in the scene 50 meters, a small shape might be hard to maneuver because the motion of the viewpoint is so fast. On the other hand, in a large scene, such motion might be scaled appropriately.

To scale the motion of the viewpoint according to the size of the scene, use the following procedure:

1. Calculate the bounding box of the scene, as follows:

   ```
   void pfuTravCalcBBox(pfNode *node, pfBox *box);
   ```

   node is generally the root node of the scene graph. *box* returns the size of the bounding box of the scene. pfBox is a structure defined as follows:

   ```
   typedef struct {
       pfVec3 min;
       pfVec3 max;
   } pfBox;
   ```

   The minimum and maximum 3D coordinates specify the lower-left, upper-right, and front and back corner of the box, respectively.

2. Adjust the speed and acceleration of the viewpoint based on the size of the bounding box, as follows:

   ```
   void pfiXformerLimits(pfiXformer* XF, float maxSpeed,
       float angularVel, float maxAccel, pfBox* dbLimits);
   ```

   This method, in pfiXformer, sets the maximum speed of *XF* to *maxSpeed*, the angular velocity of *XF* to *angularVel*, the maximum acceleration of *XF* to *maxAccel,* and the bounds within which *XF* can move to be the bounding box, *dbLimits*.

Example of Implementing User Interaction

Example 14-1 shows in bold the code used to implement transforming the view according to the motion of a mouse.

Example 14-1 Implementing User Interaction

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

```
#include <Performer/pfui.h>
#include <stdio.h>


/* Function prototypes */

void windowSetup(char *title);
void sceneSetup(char *filename);
void channelSetup(void);
void xformerSetup(void);
void handleEvents(void);
void printHelp(char *progName);


/* Global variables */

pfScene               *scene;
pfChannel             *chan;
char                  *progName;
int                   exitFlag = 0;
pfuEventStream        events;
pfuMouse              mouse;
pfiTDFXformer         *xformer;


int main(int argc, char *argv[])
{
    extern char            *progName;
    extern int             exitFlag;
    extern pfuMouse        mouse;
    extern pfiTDFXformer       *xformer;
    char             *filename = “esprit.flt”;


    /* Initialize Performer and create the pipe */

    pfInit();
    pfuInitUtil();
    pfiInit();
    pfConfig();


    /* Set up a window, scene graph, and channel */

    progName = argv[0];
```

```
    windowSetup(progName);

    if (argc >= 2) filename = argv[1];
    sceneSetup(filename);

    channelSetup();
    xformerSetup();


    /* Simulate */

    printHelp(progName);
    while ( !exitFlag) {
        pfuGetMouse(&mouse);
        pfiUpdateXformer(xformer);
        pfFrame();
        handleEvents();
    }


    /* Clean up */

    pfuExitInput();
    pfuExitUtil();
    pfExit();
    return 0;
}


void windowSetup(char *title)
{
    pfPipe                *pipe;
    pfPipeWindow          *win;

    pipe = pfGetPipe(0);
    win = pfNewPWin(pipe);
    pfPWinName(win, title);
    pfPWinSize(win, 500, 500);

    pfPWinType(win, PFPWIN_TYPE_X);
    pfuInitInput(win, PFUINPUT_X);

    pfOpenPWin(win);
}
```

```
void sceneSetup(char *filename)
{
    extern pfScene            *scene;
    pfNode                    *model;
    pfLightSource             *light;

    scene = pfNewScene();

    light = pfNewLSource();
    pfAddChild(scene, light);

    pfFilePath("/usr/people/perf/pf_data");
    model = pfdLoadFile(filename);
    pfAddChild(scene, model);
}


void channelSetup(void)
{
    extern pfScene             *scene;
    extern pfChannel           *chan;
    pfPipe                     *pipe;
    pfSphere                   bsphere;
pfEarthSky                     *esky;

    pipe = pfGetPipe(0);
    chan = pfNewChan(pipe);
    pfChanScene(chan, scene);

    pfGetNodeBSphere(scene, &bsphere);
    pfChanNearFar(chan, 1.0f, 10.0f * bsphere.radius);
    pfChanFOV(chan, 60.0f, -1.0f);

    esky = pfNewESky();
    pfESkyAttr(esky, PFES_GRND_HT, -0.1f);
    pfESkyColor(esky, PFES_GRND_NEAR, 0.0f, 0.4f, 0.0f, 1.0f);
    pfESkyColor(esky, PFES_GRND_FAR, 0.0f, 0.4f, 0.0f, 1.0f);
    pfESkyMode(esky, PFES_BUFFER_CLEAR, PFES_SKY_GRND);

    pfChanESky(chan, esky);

}
```

```
void xformerSetup(void)
{
    extern pfScene                *scene;
    extern pfChannel              *chan;
    extern pfuEventStream         events;
    extern pfiTDFXformer          *xformer;
    extern pfuMouse               mouse;
    pfCoord                       view;
    pfSphere                      bsphere;
    pfBox                         bbox;
    float                         speed;

    xformer = pfiNewTDFXformer(pfGetSharedArena());
    pfiXformerAutoInput(xformer, chan, &mouse, &events);
    pfiXformerAutoPosition(xformer, chan, NULL);
    pfiSelectXformerModel(xformer, PFITDF_FLY);

    pfGetNodeBSphere(scene, &bsphere);
    pfSetVec3(view.xyz, 0.0f, -2.0f * bsphere.radius, 1.0f);
    pfSetVec3(view.hpr, 0.0f, 0.0f, 0.0f);
    pfiXformerCoord(xformer, &view);
    pfiXformerResetCoord(xformer, &view);

    pfuTravCalcBBox(scene, &bbox);
    speed = bsphere.radius / 3.0f;
    pfiXformerLimits(xformer, speed, 90.0f, speed/2.0f, &bbox);
}


void handleEvents(void)
{
   extern pfuEventStream             events;
   extern char                      *progName;
   extern int                        exitFlag;
   extern pfiTDFXformer             *xformer;
   int                               i, j;
   int                               key, dev, val, numDevs;
   pfuEventStream                   *pEvents = &events;

   pfuGetEvents(&events);
   numDevs = pEvents->numDevs;

   for ( j=0; j < numDevs; ++j) {
      dev = pEvents->devQ[j];
      val = pEvents->devVal[j];
```

```
if ( pEvents->devCount[dev] > 0 ) {
   switch ( dev ) {

    case PFUDEV_REDRAW:
      pEvents->devCount[dev] = 0;
      break;

    case PFUDEV_WINQUIT:
      exitFlag = 1;
      pEvents->devCount[dev] = 0;
      break;

    case PFUDEV_KEYBD:
      for ( i=0; i < pEvents->numKeys; ++i ) {

         key = pEvents->keyQ[i];
         if ( pEvents->keyCount[key] ) {

            switch ( key ) {
             case 27:                  /* ESC key. Exits prog */
               exitFlag = 1;
               break;

             case 'h':
               printHelp(progName);
               break;

             case 'r':
               pfiStopXformer(xformer);
               pfiResetXformerPosition(xformer);
               break;

             default:
               break;
            }
         }
      }
      pEvents->devCount[dev] = 0;
      break;

    default:
      break;
   }
}
```

```
    }
    pEvents->numKeys = 0;
    pEvents->numDevs = 0;
}


void printHelp(char *progName)
{
    printf("\n%s - using a transforme\n\n"
        "<h> key\t\t\t- print help\n"
                  "<r> key\t\t\t- reset transformer position\n"
        "ESCAPE key\t\t- exit the program\n\n",
        progName);

}
```

Optimizing Performance

Optimizing the performance of your application is absolutely necessary to enable your images to be drawn to the buffer as quickly as possible. When your application requires too much time to render a scene, the frame rate is broken and such things as calligraphic lights are not rendered at all.

One way to optimize your application is not to draw shapes that are out-of-sight. This chapter contains the following sections, which explain how to eliminate shapes that are out of sight from the rendering.

General Performance Tips

Do not run other IrisGL or OpenGL-based applications, such as Showcase, while running OpenGL Performer applications. If you run more than one GL application on a single-pipe machine, you incur graphics context switching overhead as the applications contend for control of the pipe.

Run applications as root so that you can:

- Set nondegrading priorities
- Restrict processors

Restricting processors allows you to reduce contention for CPU time. No other processes can run on a restricted processor. See pfStageConfigFunc for an example of how to use **sysmp()** to customize each pipe stage.

Nondegrading priorities are necessary to ensure response times while an application is running. Use **schedctl()** to set nondegrading priorities.

Displaying Statistics

The statistics display shows performance information. The type of information displayed depends on the tokens passed to pfStatsClass, including:

- Time required for a frame to complete the APP, CULL, and DRAW stages.
- Load and stress.
- CPU usage.
- Rendering performance.
- Fill statistics.

You can, for example, display just one set of statistics, as shown in Figure 15-1.

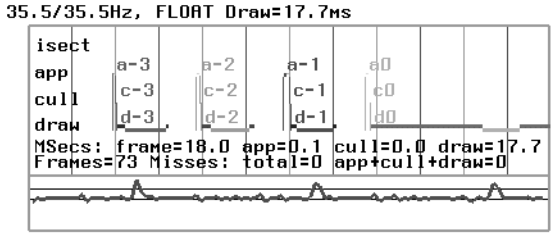

Figure 15-1 Statistics Display

Alternately, you can display many sets of statistics, as shown in Figure 15-2. The overhead for the statistical querying, however, can be expensive.

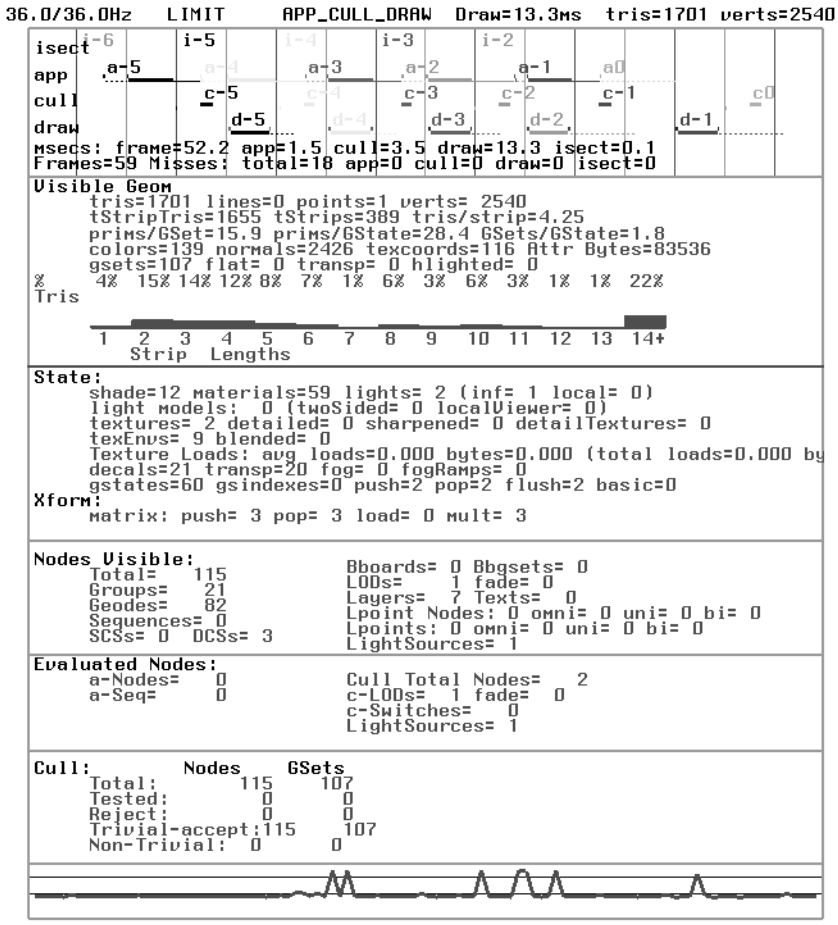

Figure 15-2 Various Statistical Modes

Rendering the Statistics Tool

To attach the statistics tool to a channel, use the following method:

```
void pfDrawChanStats(pfChannel* chan)
```

Specifying the Statistics to Gather

Because some statistics can be expensive to gather and may influence other statistics, statistics are divided into different classes, based on the tasks they monitor. You select the specific statistics of interest using pfStatsClass.

```
uint pfStatsClass(pfStats *stats, uint enmask, int val);
```

stats is the statistics class you want to enable. Valid values are the PFSTATS_ tokens listed in Table 15-1.

val specifies if the statistics class is enabled. Valid values for the PFSTATS_EN tokens are listed in Table 15-1.

Table 15-1 Statistics Class Table

Class	PFSTATS_ Token	PFSTATS_EN Token
Graphics rendered	PFSTATS_GFX	PFSTATS_ENGFX
Pixel fill	PFSTATSHW_GFXPIPE_FILL	PFSTATSHW_ENGFXPIPE_FILL
CPU	PFSTATSHW_CPU	PFSTATSHW_ENCPU
GfxPipe	PFSTATSHW_GFXPIPE_TIMES	PFSTATSHW_ENGFXPIPE_TIMES

Tokens can be ORed with other statistics-enabling tokens to enable and disable multiple statistics operations.

Statistics classes have different modes of collection so that performance-expensive modes of a particular statistics class may be disabled with pfStatsClassMode.

```
uint pfStatsClassMode(pfStats *stats, int class, uint mask, int val);
```

For a list of the mask values that specify the mode of the statistics class to use, see the *OpenGL Performer Programmer's Guide*.

Reducing Bottlenecks

The purpose of using the statistics display is to determine what stage of the rendering process requires the most time. To reduce bottlenecks, do the following:

- Make sure the best multiprocessing model is used; the stage that requires the most time should have its own process.
- If the APP stage is the bottleneck, you might have too many scene and channel changes, creating excessive CPU calculations. Use the CPU profiling tools, `prof` and `pixie` compiler tools, and `cvperf` and `cvd` from CaseVision, to find where too many calculations are being done.
- If the CULL stage is the bottleneck, compare the spatial organization of the elements in the scene to the grouping of nodes in the scene graph; the two should resemble one another.

  Use pfChanTravMode to limit culling calculations.
- The DRAW stage is the most common bottleneck. To reduce the time spent in the DRAW stage:
  - Minimize graphic state changes using **pfdMakeShared()** and **pfdMakeSharedScene()**.
  - Turn off expensive pixel operations, such as blending and multisampling.
  - Use 16-bit texel formats using pfTexFormat.
  - Buy more raster managers.
  - Reduce the LOD.
  - Use pfFlatten and pfdCleanTree to minimize transformations of static shapes.
  - Create smaller pfGeoSets or pfGeoArrays with smaller bounding boxes to allow more finely grained culls.
  - Use triangle strips instead of triangles; the longer the strip, the better.
  - Substitute billboards for complete geometries.
  - Minimize the number of active light sources.
  - Use pfGSetDrawMode to create GL display lists since they transfer to graphics pipeline efficiently.
  - Consider the topics presented in the remaining sections of this chapter.
  - To determine if the number of pixels is limiting performance, make the window smaller. If the frame rate jumps, performance is limited by the rate at which pixels are filling the polygons.

– To determine if the number of vertices is limiting performance, turn off the lighting. If the frame rate jumps, performance is limited by per-vertex calculations.

Culling Unseen Shapes

One way to increase the rendering speed of an application is to not render unseen shapes in the scene graph. OpenGL Performer provides three ways to eliminate unseen shapes from rendering calculations:

- CULL process—eliminates shapes outside the viewing frustum.
- pfCullFace—eliminates the back side of shapes, such as the rear half of a ball.
- pfBillboard—uses only a slice of a shape to represent the entire shape.

The following sections describe these OpenGL Performer features.

CULL Process

The CULL process eliminates from rendering calculations all of those shapes not in the viewing frustum. The viewing frustum is what is in the view of the channel, as illustrated in Figure 5-3 on page 82.

The CULL process checks to see if the bounding sphere of a shape is in the viewing frustum. A bounding sphere is a sphere roughly the size of the shape it encloses. A bounding sphere is used because testing a sphere is computationally less expensive than testing each point on the surface of a shape.

Evaluating Bounding Spheres

The CULL process tests the bounding spheres of shapes to see whether or not the spheres are:

- Totally inside the viewing frustum
- Totally outside the viewing frustum
- Partially inside and outside the viewing frustum

In the first two cases, the children nodes are not tested; all of the nodes are drawn or none of them are drawn, respectively.

In the last case, the children nodes are tested. All three cases are then used at each level of the subgraph.

Figure 15-3 shows each of the cases: the ball and box are totally inside or outside of the viewing frustum, respectively. The triangle is partially inside the viewing frustum.

The scene graph in Figure 15-3 shows how the CULL process eliminates nodes from rendering, according to whether or not they are visible.

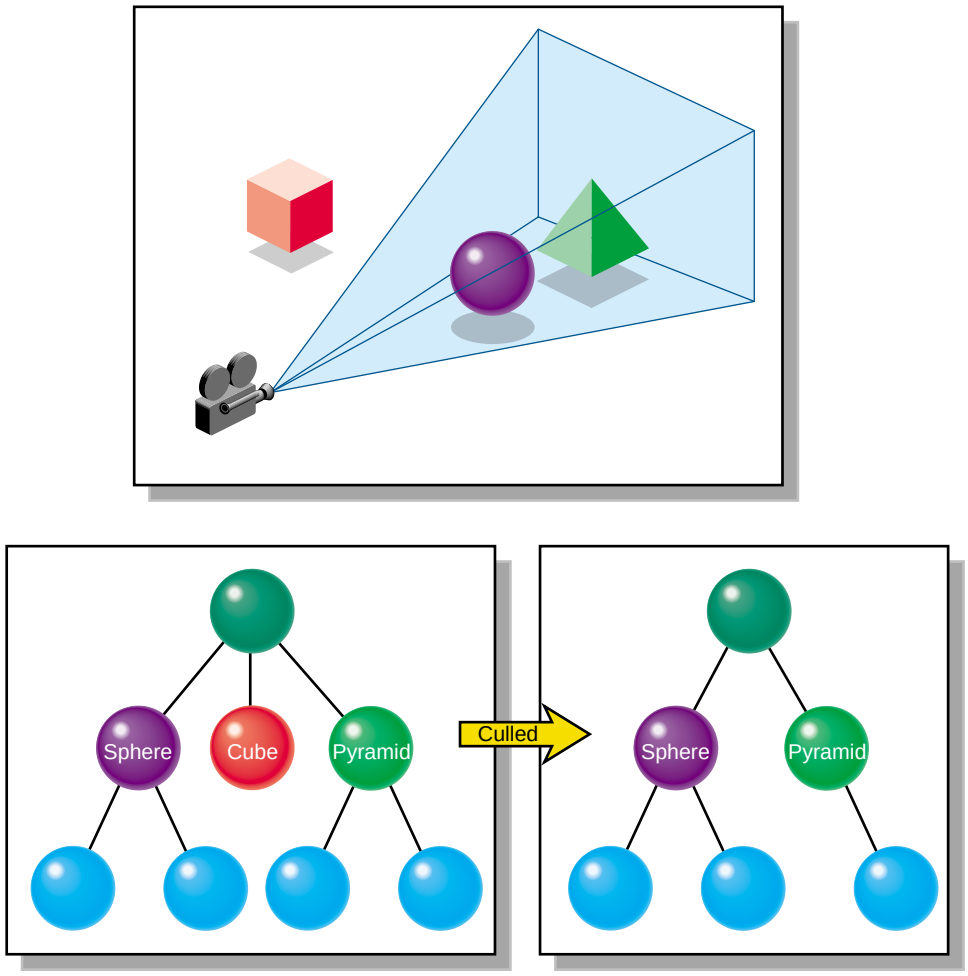

Figure 15-3 Culling Process

Optimizing the CULL Process

View frustum culling works best when:

- The objects in a pfGroup node are spatially close together, for example, all of the nodes representing a body are linearly hierarchical. When this is the case, the CULL process only needs to visit the top of the body subgraph. If the body nodes were

distributed horizontally, the CULL process would have to visit at least some of the other body nodes.

- The shapes are small compared to the full database size. If a shape is large, very likely part of it will be inside the viewing frustum so the children nodes of it must be tested, which hinders performance.

  Objects that are roughly the same length in each of the three dimensions cull better than long, thin objects. An object that spans the database, for example, a beam across the ceiling of a building, cannot be culled as easily as two halves of the beam. It may be useful to divide objects that can be easily divided.

pfdGeoBuilder provides tools to group together in the scene those graph nodes whose shapes are close together in world space. OpenGL *Optimizer* also provides tools for arranging scene graph nodes spatially, as well as tools for creating LOD children.

Face Culling

When a three-dimensional shape is rendered, the side of it facing away from the camera is normally hidden by the side that faces the camera. For example, when a sphere is rendered, you normally only see its front side. You can avoid rendering the back side of a shape using **pfCullFace()**. Backface culling is enabled by default in pfPipeWindows for `libpf` applications.

The **pfCullFace()** mode specifies how much of a shape is rendered. The possible values include:

- PFCF_OFF—Both the front and back sides of shapes are rendered.
- PFCF_FRONT—Only the back sides of all shapes are rendered.
- PFCF_BACK—Only the front sides of all shapes are rendered.
- PFCF_BOTH—Shapes are not rendered.

pfGetCullFace() returns one of these values, whichever is current.

Not rendering either the front or back side of a geometry improves rendering performance.

Rendering Slices of Shapes

Some shapes are symmetric in the horizontal plane and vertical planes, such as a sphere. Other shapes are roughly symmetric around one axis, (for example, a tree is generally symmetric around the z-axis).

Rather than render the complete shape in great detail, pfBillboard rotates a slice of a shape so that it always faces the camera. In this way, if you move a camera around a tree, the same pfBillboard slice of the image revolves around the location of the tree such that the slice appears to be the tree. The tree appears to lack the specificity of a fully rendered shape because it appears the same from all sides; on the other hand, not rendering the entire tree in detail increases the performance of the application.

Rotating the Slice

A pfBillboard can rotate freely around a point or it can rotate around an axis. pfBillboard objects approximating shapes symmetric around two axes should use the point mode. pfBillboard groups approximating shapes symmetric around one axis should use the axis mode.

To specify the mode of rotation, use one of the following tokens as the value for *mode* in the argument of **pfBboardMode()**:

- PFBB_POINT_ROT_EYE—to rotate the billboard around a point.
- PFBB_AXIAL_ROT—to rotate the billboard around an axis.

Maintaining Frame Rate Using DVR

When there is too much data to render, the frame is not updated when the frame is refreshed. The result of inconsistent frame rates is jerky motion within the scene.

The key to maintaining frame rate is limiting the amount of information to be rendered. OpenGL Performer can take care of this problem automatically when you use the PFPVC_DVR_AUTO token with **pfPVChanDVRMode()**. This mode is called Dynamic Video Resolution (DVR).

In PFPVC_DVR_AUTO mode, OpenGL Performer checks every rendered frame to find out if it required too much time to render. If it did, OpenGL Performer reduces the size of the image, and correspondingly, the number of pixels in it. Afterwards, the video

hardware enlarges the images to the same size as the pfChannel; in this way, the image is the correct size, but it contains a reduced number of pixels, as suggested in Figure 15-4.

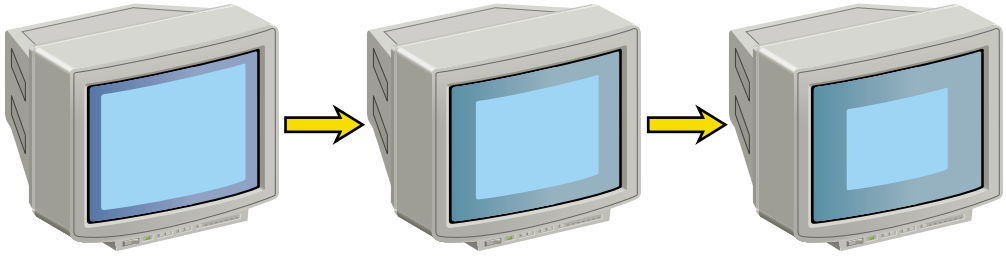

Figure 15-4 Real Size of Viewport Rendered under Increasing Stress

Although the viewport is reduced as stress increases, the viewer never sees the image grow smaller because bipolar filtering is used to enlarge the image to the size of the channel.

DVR Scaling

DVR scales linearly in response to the most common cause of draw overload: filling the polygons. For example, if the DRAW stage process overruns by 50%, in order to get back in under the time frame, the new scene must reduce the dimensions of the viewport by 30% in both dimensions because 0.7 X 0.7 = 0.49; (almost a 50% reduction in the number of pixels drawn.)

DVR can automatically render to a smaller viewport and let the video hardware rescale the image to the correct display size.

If pfPVChanMode is DVR_AUTO, OpenGL Performer automatically scales each of the pfChannels. pfChannels automatically scale themselves according to the scale set on the pfPipeVideoChannel they are using.

If the pfPVChanMode is DVR_MANUAL, you control scaling according to your own policy by setting the scale and size of the pfPipeVideoChannel in the application process between **pfSync()** and **pfFrame()**.

Note: For more information about customizing DVR or understanding the stress filter used by DVR, see Chapter 5, "Frame and Load Control," in the *OpenGL Performer Programmer's Guide*.

Level of Detail Reduced for Performance

The children of a level of detail (pfLOD) node each encapsulate a shape at a different level of resolution. The factor of resolution between children of a pfLOD is often one quarter; so when a lower resolution child replaces the current pfLOD child displayed, only one quarter of the current number of vertices needs to be rendered. The maximum reduction of detail is when all of the vertices of the highest-resolution image are reduced to a single pixel.

The pfLOD (level of detail) node is a subclass of pfSwitch. pfLOD switches between its child nodes, based on the proximity of an object to the camera.The further a shape is from the viewer, the less resolution is needed to display it. OpenGL Performer switches between the children automatically, based on a range value, to display a shape at the correct level of resolution.

pfLOD allows you to reach a compromise between performance and the level of detail rendered. For high quality images, a shape close to the camera should be rendered in high detail. When a shape recedes from the camera, the same level of detail is not necessary. Reducing the level of image detail reduces the number of vertices required to render a shape, which results in improved performance.

OpenGL Optimizer can create the pfLOD children nodes.

Choosing a Child Node Based on Range

Each child node of a pfLOD node is associated with a range. The range can be defined as the distance over which a child of the pfLOD is displayed from the camera, expressed in world space.

Shapes are not displayed if:

- They are closer to the camera than the beginning distance of the closest (highest resolution) LOD child.

- They are further away than the farthest distance of the farthest (lowest resolution) LOD child.

The distance between the camera and the shape is computed during the traversal of the scene graph and the correct LOD child is automatically displayed.

Setting the Range

You set the range value using the following pfLOD methods:

```
void pfLODCenter(pfLOD *lod, pfVec3 center);
void pfLODCenter(const pfLOD *lod, pfVec3 center);

void pfLODRange(pfLOD *lod, int index, float range);

void pfLODTransition(pfLOD *lod, int index, float distance);
int pfGetLODNumRanges(const pfLOD *lod);
```

The **pfLODCenter()** method specifies the *center* of the LOD. The range over which a particular LOD child node is displayed is calculated as the *center,* plus or minus the *range* value specified in **pfLODRange()**.

pfLODRange() associates the child LOD node with its range. The child node is identified by its index number, *index*, where the highest resolution node is index number zero.

Generally, you set up a loop to specify the range values for the child pfLOD nodes, using the returned value of **pfGetLODNumRanges()** as the bounding value for the number of loops.

Disregarding LODs

OpenGL Performer may disregard range values and perform as follows:

- Display an already fetched level of detail while a higher level of detail is downloaded from disk.
- Adjust the level of detail displayed to maintain a constant frame rate; this is always the case if you leave the **range()** field empty.
- Disregard the range values for any other implementation-dependent reason.

Tip: For best results, specify ranges only where necessary; give browsers as much freedom as possible to choose levels of detail based on performance.

Transitioning Between Levels of Detail

The default transition between LOD children is simply a switch from one LOD child to another. You can, however, specify a fade between LOD children over a range.

Note: To use the fade option, your platform must have multisampling hardware.

The pfLODTransition method specifies the distance over which one pfLOD child fades into the next, as shown in Figure 15-5.

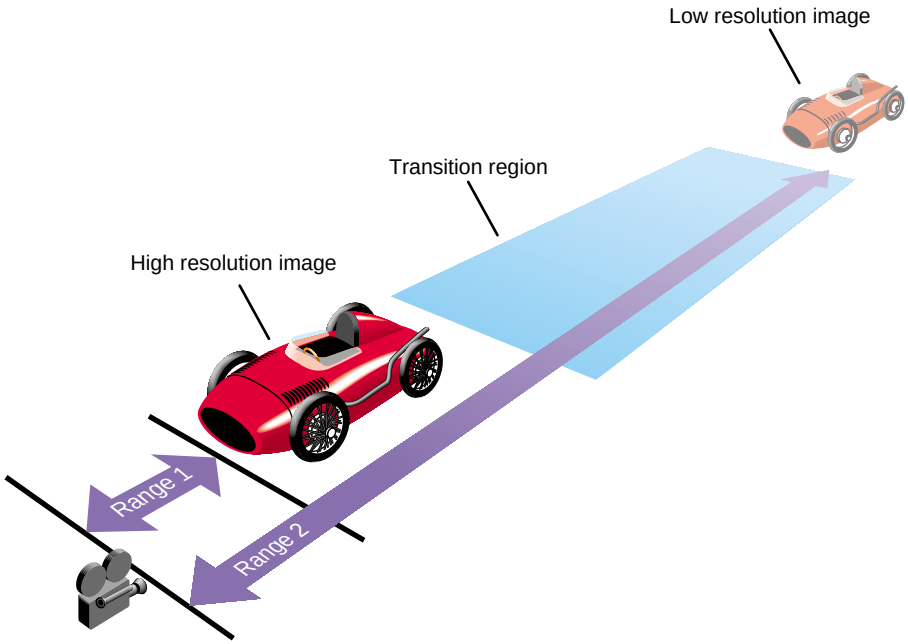

Figure 15-5 pfLOD Ranges

The *distance* value in **pfLODTransition()** is applied before and after the boundary between two LOD children, such that the fade between one LOD child to another actually occurs over 2 X *distance* value. The default value of *distance* is 1.

Fading involves an alpha blending between two LOD children, such that as one LOD fades into transparency, the other LOD becomes 100% opaque.

The drawback for fading is that both LOD children must be drawn, which hinders performance.

Enabling Fading

Even though you set the transition range in**pfLODTransition()**, the fade is not enabled by default. To enable fading between LOD children, you must set the attribute, PFLOD_FADE, in **pfChanLODAttr()**, to a non-zero value. The default value for PFLOD_FADE is 0.0: no fade. For more information about **pfChanLODAttr()**, see "Customizing LOD Actions" on page 218.

When computing the actual distance over which one LOD child fades into another, the value supplied for PFLOD_FADE is multiplied by distance values specified in **pfLODTransition()**.

Customizing the Fade

OpenGL Performer fades one LOD child into another evenly, such that at the boundary between two LOD children, both LOD children are 50% transparent and 50% opaque.

You can, however, specify an uneven rate of fading between LOD children using **pfEvaluateLOD()**. By specifying a value of 1.25 in this method, for example, at the boundary between two LOD children, the higher-resolution child would only be 25% transparent, and the other child only 75% opaque.

Similarly, a value of 3.9 would mean that the higher-resolution child would only be 10% transparent, and the other child only 90% opaque, at the boundary between two LOD children.

Customizing LOD Actions

OpenGL Performer allows you to customize LOD actions using the following pfChannel method:

```
extern void pfChanLODAttr(pfChannel* _chan, int  attr, float val);
```

attr specifies the customization; it can be one of the following values:

- PFLOD_FADE—fades between LOD child nodes.
- PFLOD_SCALE—globally increases or decrease the ranges for all LODs in a channel or channel group.
- PFLOD_STRESS_PIX_LIMIT—prevents lower-resolution LODs from displaying as a result of stress.
- PFLOD_FRUST_SCALE—changes LODs based on the size of the viewport.

val is the value you give to the *attr* argument.

These attributes are shared by all channels in a channel group. If you want to specify attributes per channel for channels in a group, use pfChanShare.

For more information about PFLOD_FADE, see "Enabling Fading" on page 217.

The following sections describe the other attribute values.

Scaling LOD Ranges

When you change the scale of the images displayed, you also need to change the scale of the LOD children ranges. The PFLOD_SCALE attribute enables you to make global changes to all of the ranges in a channel.

When you specify PFLOD_SCALE, the float you supply as its value in **pfChanLODAttr()** becomes the multiplier for all of the range values. For example, if you specify 2.0 as the value for PFLOD_SCALE, all ranges are doubled. The default value is 1.0.

This attribute is valuable for global changes in ranges.

Overriding Stress Effects

When the images displayed are complex enough that the frame rate is not maintained, simpler LODs are drawn to reduce the graphics load. In some situations, however, it is undesirable to use low-resolution LOD nodes, for example, when the shapes in the LOD node are close to the viewer and occupy considerable screen space. You can avoid this problem by using the PFLOD_STRESS_PIX_LIMIT attribute, which will keep exempt specific LODs from being affected by system stress.

When you specify PFLOD_STRESS_PIX_LIMIT, the pixel size you supply as its value in **pfChanLODAttr()** becomes the determining factor as to whether or not stress can change the selection of LOD children displayed.

Stress, computed with **pfChanStress()**, can automatically reduce the level of detail displayed if the frame rate is not maintained. The PFLOD_STRESS_PIX_LIMIT attribute specifies the pixel size, above which stress has no effect on the selection of LOD children displayed.

When the value of PFLOD_STRESS_PIX_LIMIT is less than or equal to zero, stress has no effect on the selection of LOD children displayed.

For more information about stress, see “Reducing System Stress” on page 220.

Selecting LODs Based on Viewport

When you change the size of the viewport in which a channel is displayed, you scale the image to fit the viewport. When you make the viewport smaller, the level of resolution necessary to display the images is lowered. You can reduce the graphics load by tying the scaling of the viewport with the selection of LOD children displayed, by using the PFLOD_FRUST_SCALE attribute.

When you specify PFLOD_FRUST_SCALE and supply a non-zero value for it, LOD ranges are multiplied by a factor according to the size of the viewport:

- The smaller the viewport, the shorter the ranges.
- The larger the viewport, the longer the ranges.

The general effect of the attribute is that as you reduce the size of the viewport, lower-resolution LODs are displayed.

Reducing System Stress

OpenGL Performer tries to maintain a fixed frame rate by displaying different LOD children to reduce or increase the graphics load. At the end of each frame, OpenGL Performer computes a *load* value for each pfChannel based on the length of time required to render the pfChannel. When rendering time approaches or exceeds a frame period, the stress value is increased and lower-resolution LOD children are displayed as a result.

Load is the rendering time divided by the desired frame period. The value of *stress* varies directly with the load; the more complex the display, the higher the system stress.

Setting the Stress Filter

The stress filter monitors the system load and either raises or lowers the stress value according to its parameters. OpenGL Performer multiplies the stress value times the LOD ranges for the next frame. A stress value greater than one increases the LOD ranges so that coarser LOD children are drawn, and the graphics load is reduced. If stress = 1.0, the system is not in stress and LOD ranges are not modified.

Whether or not the stress value is modified depends on the parameters in the stress filter:

```
void pfChanStressFilter(pfChannel *chan, float frac, float low,
    float high, float scale, float max);

void pfChanStress(pfChannel *chan, float stress);
```

low and *high* define a hysteresis band for system load. When the load is:

- Less than *low*, the stress value is reduced.
- Higher than *high*, the stress value is increased.
- Between *low* and *high*, the stress value is unchanged.

Stress values are clamped to the range [1.0, *max*].

Stress Volatility

Because the stress is computed for every frame, stress values can change often. An undesirable side effect in changing the stress value so often is overcompensating for too much or too little system load. For example, the stress filter could change the load, so that

in adjacent frames the system load is too great, too little, too great, too little, and so on. The effect is to display different LOD children in every frame.

To counteract this overcompensation, the stress filter includes the argument, *scale*. Stress increases or decreases proportionally to the value of scale. With a small *scale* value, stress values can change slowly; with a high *scale* value, stress values can change quickly.

Dividing Rendering Time

When rendering multiple channels, each channel must be rendered in a fraction of the frame rate. Because load is partially based on the time you expect a channel to be rendered, you can set *frac* to different values for different channels. *frac* is the proportion of time you expect it will take a channel to be rendered.

When rendering a single channel on a pfPipe, *frac* should be 1.0, because the single channel consumes all of the rendering time.

When rendering multiple channels, set the *frac* value larger for those channels that require more time to render. For example, if channel 0 showed a scene, and channel 1 showed a smaller view of channel 0 with cross hairs superimposed on it, you might set the *frac* for channel 0 to 0.7 and the *frac* for channel 1 to 0.3, because the smaller view requires less time to render.

Setting the Stress Value Explicitly

An application may set the stress filter explicitly by calling **pfChanStress()**. Stress values set by **pfChanStress()** override stress values computed by the stress filter.

Optimizing pfGeoSet Performance

Transferring geometry data from disk to system memory is time consuming. You can eliminate this delay for geometries that do not change vertices, color, normals, or texture coordinates.

The following method compiles a GL display list, which contains geometry data. The compilation of the data eliminates the data download time.

```
void pfGSetDrawMode(pfGeoState *gset, PFGS_COMPILE_GL, PF_ON);
```

A GL display list is not modifiable. You can, however, use pfDCS and pfSCS nodes to transform geometries stored in GL display lists. Another option is to use packed attribute arrays for pfGeoSet vertex attributes. See Chapter 8, “Geometry”, in the *OpenGL Performer Programmer's Guide* for more information on these topics.

Optimizing Graphics State Changes

The following tools can optimize graphic state changes:

- pfdMakeShared()
- pfdMakeSharedScene()

Sharing Common pfGeoStates

The **pfdMakeShared()** method performs as follows:

1. Finds all pfGeoStates that are the same.
2. Points all pfGeoSets or pfGeoArrays using identical pfGeoStates at the same pfGeoState object.
3. Eliminates all duplicate pfGeoState objects.

Eliminating pfGeoState objects reduces memory consumption.

Computing the Optimal, Global Graphics State

The **pfdMakeSharedScene()** method performs as follows:

1. Examines all pfGeoStates in a scene graph.
2. Computes the optimal pfGeoState values that reflect how most of the geometries look.
3. Change local pfGeoState values according to the newly calculated, global pfGeoState values.

Optimizing Texture Handling

The following tips provide improved application performance when handling textures:

- Use images with dimensions that are powers of two.
- Use 16-bit texel formats.
- Larger texels slow down the application linearly, for example, a 32-bit texel texture is twice as slow as a 16-bit texel texture.
- Download textures to hardware before the simulation loop.

For more information about downloading textures, see “Preloading Textures” on page 134.

Optimizing File Loading

Although you can use files in many formats (specified by their file extensions), you can dramatically reduce database loading time by preconverting databases into the PFB format and images into the PFI format.

To convert to the PFB file format or the PFI image format, use the `pfconv` and `pficonv` utilities.

pfconv

The `pfconv` utility converts from any format for which a **pfdLoadFile...()** function exists into any format for which a **pfdStoreFile...()** exists. The most common format to convert to is the PFB format. For example, to convert `cow.obj` into the PFB format, use the following command:

```
% pfconv cow.obj cow.pfb
```

By default, `pfconv` optimizes the scene graph when performing the conversion. The optimizations are controlled with the -o and -O command line options. Builder options are controlled with the -b and -B command line options. Converter modes are controlled with the -m and -M command line options. See the help page for more specific information about the command line options by entering:

```
% pfconv -h
```

When converting to the PFB format, texture files can be converted to the PFI format using the following command line options:

```
% pfconv -M pfb, 5, 1
```

5 means PFPFB_SAVE_TEXTURE_PFI.

1 means convert .`rgb` texture images to .`pfi`.

pficonv

The `pficonv` utility converts from the IRIS RGB image format to the PFI image format. For example, to convert `cafe.rgb` into the PFI format, use the following command:

```
% pficonv cafe.rgb cafe.pfi
```

Mipmaps can be automatically generated and stored in the resulting PFI files by adding -m to the command line.

Building a Visual Simulation Application Using libpf

This appendix 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 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.

For a more modular approach using a graphical viewer, see Appendix B, "Building a Visual Simulation Application Using libpfv".

Overview

It takes only a few lines of code to set up an OpenGL 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 A-1. If you want to compile this program, see the section "Compiling and Linking OpenGL Performer Applications" on page 237.

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 reviewing more advanced subjects.

Example A-1 shows the basic framework of an OpenGL Performer application.

Example A-1 Structure of an OpenGL 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, “OpenGL 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 want to compile `simple.c` and try it, use the copy in `/usr/share/Performer/src/pguide/libpf/C` (for IRIX and Linux systems) and `%PFROOT%\Src\pguide\lippf\C` (for Windows systems). File `Makefile` in that directory provides all the necessary compilation options. (For more information about OpenGL Performer compiler options, see section "Compiling and Linking OpenGL Performer Applications" on page 237.) Once you have compiled the code, try executing it with some of the sample data files in `/usr/share/Performer/data` (for IRIX and Linux systems) and `%PFROOT%\Data` (for Windows systems), such as `blimp.flt` or `sampler.nff`.

The following describes the steps involved in a simple OpenGL Performer application. Refer to the sample code in Example A-1 as you read these steps.

1. Include the necessary system header files.

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

2. Include the relevant OpenGL Performer header files.

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

3. Declare variables for the required elements.

pfScene	Scene graph to be rendered on a channel
pfPipe	Graphics pipeline to perform the rendering
pfChannel	View to be rendered on the designated pipe

You can configure OpenGL Performer to use multiple scenes, multiple pipes (if your system has them), and multiple channels per pipe. (See "Using Multiple Channels" on page 87.)

4. Initialize OpenGL Performer.

   ```
   pfInit();
   ```

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

5. Configure OpenGL Performer.

   ```
   pfConfig();
   ```

   This configures the number of pipes and starts processes based on the selected multiprocessing model. The code in Example A-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 example, 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 Chapter 17, "Math Routines," in the *OpenGL Performer Programmer's Guide* for information on constructing pfGeoSets, and Chapter 6,"Creating Scene Graphs," 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 with a custom window.

   ```
   p = pfGetPipe(0);
   pw = pfNewPWin(p);
   pfPWinType(pw, PFPWIN_TYPE_X);
   pfPWinName(pw, "OpenGL Performer");
   pfPWinOriginSize(pw, 0, 0, 500, 500);
   /* set up configuration callback OpenPipeWin() */
   pfPWinConfigFunc(pw, OpenPipeWin);
   ```

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

This sets up an optional callback to open a graphics library window for custom initialization, in this case, **OpenPipeWin()**. In the `simple.c` example, no window configuration callback was used.

10. Specify the frame rate and the synchronization method.

```
pfFrameRate(60);
pfPhase(PFPHASE_LOCK);
```

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 234 and Chapter 10, "Controlling Frame Rate," 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—OpenGL Performer matches the FOV to the aspect ratio of the channel.

13. Render the scene repeatedly until the specified time has elapsed.

- 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 limit is reached, exit OpenGL Performer.

```
pfExit();
```

The remainder of this appendix 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 OpenGL Performer `libpf` application.

Using OpenGL Performer Header Files

The header files for the OpenGL Performer libraries are in the `/usr/include/Performer` directory (for IRIX and Linux systems) and `%PFROOT%\Include\Performer` (for Windows systems). They include `pf.h` and `pr.h` (header files for `libpf` and `libpr`, respectively), and `opengl.h` and other header files for use with the other OpenGL 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 OpenGL 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

OpenGL 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 Chapter 5, "Frame and Load Control," in the *OpenGL Performer Programmer's Guide*.

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 "Multiple Pipes" on page 91. For information on multiprocessing, see Chapter 5, Frame and Load Control," and Chapter 20, "Programming with C++," in the *OpenGL Performer Programmer's Guide*.

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 OpenGL Performer software graphics pipeline. You gain access to a pipe using **pfGetPipe()**, for example,

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

This statement sets *p* to point to the OpenGL Performer graphics pipeline numbered zero. The optional function **pfStageConfigFunc()** function for the PFPROC_DRAW stage sets up a callback initializes the drawing process for the pfPipe. **pfWinConfigFunc()** sets up a callback to do custom window initialization, as shown in Example A-2.

Example A-2 pfStageConfigFunc() Callback

```
/* Set up pipe config func. */
pfStageConfigFunc(0, PFPROC_DRAW, ConfigPipe);
/* Set up window config func. */
pfPWinConfigFunc(pw, OpenPipeline);
/* Trigger pipe stage config func in draw process. */
pfConfigStage(0, PFPROC_DRAW);
/* Trigger window config func in draw process. */
pfConfigPWin(pw);
```

The first argument to **pfStageConfigFunc()** is the pipe number and (-1) will select all pipes. The next argument is the stage or process in which you select the DRAW. The third argument is the configuration function.You perform custom window initialization in the Window configuration function, **OpenPipeline()**. If there is a custom window configuration function, it must open the window with **pfOpenWin()** as shown in Example A-3.

Example A-3 Sample OpenPipeline() Routine

```
void OpenPipeline(pfPipe *p)
{
    /* Open the window. */
pfOpenWin();

    /* initialize custom graphics state */
pfCullFace(PFCF_BACK);
}
```

Whether or not you specify a function with **pfConfigPWin()**, a custom window (as opposed to the default full screen window OpenGL Performer will otherwise create and open for you automatically) is not opened until a call to **pfOpenWin()** is made.

The call to **pfInitGfx()** sets up the initial graphics library state for OpenGL Performer and is called automatically by **pfOpenWin()**. You can also call this to re-initialize your window. The graphics library maintains state information for graphics hardware and software settings. These settings, or modes, determine how graphics are processed and rendered. Because OpenGL Performer takes over the processing and rendering duties of the system, the system must be set to a known state before it can reliably proceed.

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

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` does not 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 do not require a fixed frame rate. **pfSync()** delays the application until the next appropriate frame boundary.

See Chapter 10,“Controlling Frame Rate," 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 Chapter 5, “Creating a Display with pfChannel," to learn more about setting up channels.

Creating and Loading a Scene Graph

Databases exist in a variety of formats. OpenGL Performer does not 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 OpenGL Performer; the provided importers are described in Chapter 7, “Importing Databases”, in the *OpenGL Performer Programmer's Guide*.

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 7, “Importing Databases”, in the *OpenGL Performer Programmer's Guide* for more information about import routines.

If you write your own modeler using `libpr` routines, you do not 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 or pfGeoArrays, by defining state and texture definitions in pfGeoStates, and by constructing a scene graph of pfNodes.

Setting the Search Path for Database Files

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

When OpenGL 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***)**, **pfFilePathv(***path0*, *path1*, ..., *pathn*, NULL**)**, or with the environment variable PFPATH. You can specify any number of directories using **pfFilePath()** and a maximum of 64 using **pfFilePathv()**. Colons separate path names on IRIX and Linux and semicolons on Windows. Since

pfFilePathv() allows you to specify path names delimited by commas, it provides much more economy in coding compared to the use of **pfFilePath()**, where you must employ conditional code to accomodate cross-platform use. Directories are searched in the order given, beginning with those specified in PFPATH, followed by those specified by **pfFilePath()** or **pfFilePathv()**.

For example, the following function call tells OpenGL 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, and then in the installed OpenGL Performer data directory:

```
pfFilePath(".",
           "./data",
           "../data",
           "../../data",
           "/usr/share/Performer/data",
            NULL);
```

If you call **pfFindFile()** with the name of the file you want to locate, the complete pathname of the file is returned 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 used only 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 appendix does not specifically discuss performance tuning (see Chapter 15,“Optimizing Performance,"), but every OpenGL Performer-based application should be written with performance in mind. You cannot easily build speed into an application as a last-minute addendum. During the design of your program (rather than after debugging it), you need to consider speed as you structure your database, as you decide what needs to happen in your main loop, and so on.

Compiling and Linking OpenGL Performer Applications

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

Required Libraries

The following libraries are required when linking an executable on the IRIX and Linux operating systems:

`libpf`	OpenGL Performer visual simulation development library.
`libpr`	OpenGL Performer high-performance rendering library—exists in OpenGL and is contained within the corresponding `libpf`.
`libpfdu`	OpenGL Performer database library— does file handling, and includes importers for a variety of data formats.
`libpfutil`	OpenGL Performer utilities library— includes the window-related functions.
`libimage`	Image library—required by `libpr`.
`libGLU`	OpenGL utilities library—required by `libpr` with `OpenGL.`
`libGL`	OpenGL graphics library—is required by `libpf` and `libpr`.

`libXsgivc`	X extensions—for video control.
`libXext`	X extensions—needed by Silicon Graphics X extensions libraries.
`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 for Motif.
`libXmu`	X utility library required by libpr.
`libX11`	X Window System library—required by `libgl` and `libpr`.
`libm`	Math library—required by `libpr`.
`libfpe`	Floating point exception library—required by `libpr`.
`libC`	C++ library—required by `libpf`.

The corresponding line for an OpenGL application would be the following:

```
-lpfdu -lpfui -lpfutil -lpf -limage -lGLU -lGL -lXext -lXsgivc -lXmu
-lX11 -lm -lfpe -lC
```

The following libraries are required when linking an executable on the Windows operating systems:

`libpf.lib`

`libpr.lib`

`libpfdu-util.lib`

`glu32.lib`

`opengl32.lib`

`gdi32.lib`

`user32.lib`

`msvcrt.lib` Multithreaded Optimized C library

The corresponding line for an OpenGL Performer application would be the following:

```
libpfdu-util.lib libpfui.lib libpf.lib glu32.lib opengl32.lib
gdi32.lib user32.lib /NODEFAULTLIB:LIBC /NODEFAULTLIB:MSVCRT
/NODEFAULTLIB:MSVCPRT msvcprt.lib msvcrt.lib
```

Dynamic Shared Objects (DSOs)

The standard libraries for OpenGL Performer are distributed as *dynamic shared objects (DSOs)*. 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 OpenGL Performer 3.0 are in the performer_eoe subsystem on the OpenGL Performer CD-ROM. OpenGL Performer 3.0 on IRIX also includes DSOs from all previous versions in compatibility subsystems, so that old OpenGL Performer programs will still run on a system with OpenGL Performer 3.0 installed.

Note: On Windows systems, a dynamic link library (DLL) is the functional equivalent of a DSO.

Debug and Static Libraries

OpenGL Performer is also shipped with 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 the *OpenGL Performer Programmer’s Guide* for information concerning file readers, which are normally accessed as DSOs at run time even when the main OpenGL Performer libraries have been statically linked. Also, when linking statically, you will have to be sure that you have all required libraries on your link line.

Static libraries are not available for Windows systems.

Using Compiler Flags

Note: This section is not applicable to Windows systems.

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 OpenGL Performer in the following ways:

1. Because `-cckr` does not 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` does not 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 Performer applications in 64-bit mode.

To do this, you need to have installed the optional 64-bit versions of the OpenGL 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 OpenGL 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 Power Onyx 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 OpenGL Performer and use the "-n32" option to the compiler. The old 32-bit, new 32-bit and 64-bit versions of OpenGL 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.

The sample Makefiles in the source code distribution recognize the environment variable PFSTYLE and values of 32 for o32, N32 for n32, and 64 for 64-bit compilation.

Using OpenGL Performer From C++

OpenGL 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 OpenGL 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 OpenGL Performer programs using the C++ API, see the following directories for examples of C and C++ programs:

`/usr/share/Performer/src/pguide`
`/usr/share/Performer/src/apps`
(IRIX and Linux)

`%PFROOT%\Src\pguide`
`%PFROOT%\Src\apps`
(Windows)

Building a Visual Simulation Application Using libpfv

In contrast to Appendix A, "Building a Visual Simulation Application Using libpf", this appendix describes how to use the library `libpfv` to build an application using a graphical viewer.

This appendix has the following sections:

Overview

OpenGL Performer includes `libpfv`, a C++ library for easy construction of modular, interactive OpenGL Performer applications.

The library `libpfv` supports the following features:

- Reading and writing XML files
- Specifying complex display configuration (pipes, windows, and channels) from a file or through API calls

- Tracking mouse and keyboard input
- Setting up user interaction with 3D scene elements
- Managing multiple scene graphs (worlds)
- Managing multiple camera positions (views)
- Extending program functionality using program modules

The principal class in `libpfv` is pfvViewer. It allows complex multiworld and multiview applications to be implemented in a modular fashion, allowing individual features to be encapsulated into configurable and re-usable modules.

In addition to `libpfv`, OpenGL Performer includes ready-to-use modules that provide the following features:

- Loading geometry into a pfvViewer world
- Picking geometry under the mouse pointer
- Manipulating geometry (rotating, translating, scaling, deleting)
- Navigating through a world using mouse and keyboard controls
- Controlling the render style of models
- Setting up colorful earth and sky backgrounds
- Displaying 2D images in overlay
- Saving snapshots of the rendered images
- Smoothly transitioning from one world to another
- Collecting and displaying scene graph statistics

The Simplest pfvViewer Program

The pfvViewer class starts with a very simple programming interface. It maintains a very simple programming interface even when accessing high-level features. As shown in the following program, the simplest program using pfvViewer loads a model and places the camera at a comfortable viewing distance:

```
#include <Performer/pfdu.h>
#include <Performer/pf/pfLightSource.h>
#include <Performer/pfv/pfvViewer.h>
```

```
main (int argc, char *argv[])
{
    // Initialize Performer
    pfInit();

    // Create a new pfvViewer
    pfvViewer* viewer = new pfvViewer;

    // Initialize loading of a model file.
    pfdInitConverter(argv[1]);

    // Configure/Initialize pfvViewer
    viewer->config();

    // Add a light source to the world.
    viewer->addNode(new pfLightSource);

    // Add a model to the world
    viewer->addNode(pfdLoadFile(argv[1]));

    // Start viewing
    viewer->run();
}
```

Adding Interaction to a pfvViewer Program

To add interaction, load the following two standard modules into the viewer:

- `pfvmNavigator` module

  Allows the user to move around the 3D scene through mouse and keyboard input.

- `pfvmPicker` module

  Allows the user to select, manipulate, and delete portions of the 3D scene through mouse and keyboard input.

The following program shows the addition of these modules to the simple program in the preceding subsection:

```
#include <Performer/pfdu.h>
#include <Performer/pf/pfLightSource.h>
#include <Performer/pfv/pfvViewer.h>
```

```
main (int argc, char *argv[])
{
    pfvModule* module;

    // Initialize Performer
    pfInit();

    // Create a new pfvViewer
    pfvViewer* viewer = new pfvViewer;

    // Add navigation module
    module = pfvModule::load("pfvmNavigator");
    viewer->addModule(module);

    // Add mouse-picking module
    module = pfvModule::load("pfvmPicker");
    viewer->addModule(module);

    // Initialize loading of model files.
    pfdInitConverter(argv[1]);

    // Configure/Initialize pfvViewer
    viewer->config();

    // Add a light source to the world.
    viewer->addNode(new pfLightSource);

    // Add a model to the world
    viewer->addNode(pfdLoadFile(argv[1]));

    // Start interaction
    viewer->run();
}
```

Reading XML Configuration Files

A pfvViewer can read most of its parameters from an XML configuration file. A pfvViewer configuration file, denoted by the `.pfv` extension, can contain any of the following items:

- Display configuration (pipes, windows, and channels)

- Specification of multiple worlds
- Specification of multiple views (camera positions)
- Extension modules to be loaded into the pfvViewer and their specific configuration parameters

The simplest pfvViewer program using an XML configuration file is `pfview`, which is provided as a precompiled executable with OpenGL Performer.

The source code for the `pfview` program looks like the following:

```
#include <Performer/pfv/pfvViewer.h>

int
main (int argc, char *argv[])
{
    // Initialize Performer
    pfInit();

    // Create a new pfvViewer and read XML configuration file argv[1]
    pfvViewer* viewer = new pfvViewer(argv[1]);

    // Configure/Initialize pfvViewer.
    viewer->config();

    // Start interaction
    viewer->run();
}
```

A minimal XML configuration file suitable for `pfview` has the following structure:

```
<?xml version="1.0" ?>
<viewer>
    <module>
        <!-- Loader module: loads models into world -->
        <class>pfvmLoader</class>
        <data>
            <Model>
                <FileName>esprit.flt</FileName>
            </Model>
        </data>
    </module>

    <!-- Add a trackball navigation module -->
    <module>
```

```
        <class>pfvmTrackball</class>
    </module>

    <!-- Add a picking module -->
    <module>
        <class>pfvmPicker</class>
    </module>

</viewer>
```

You can use a more complex XML configuration file, one including a `display` tag, to set up complex display configurations. The following file is an example that sets up a panoramic view over three channels, each rendered in a separate graphics pipe:

```
<?xml version="1.0" ?>
<viewer>
    <!-- Display specifications -->
    <display>
        <!-- Configure middle pipe -->
        <pipe>
            <!-- direct middle pipe to screen 0 -->
            <screen>0</screen>
            <!-- Configure a single pipe-window on middle pipe -->
            <pwin>
                <!-- set pipe-window to fullscreen, no border -->
                <fullscreen>1</fullscreen>
                <border>0</border>
                <!-- Configure a single channel on pipe-window -->
                <chan>
                    <viewrange>0.32,0.68,0.0,1.0</viewrange>
                    <hprOffset>0.0,0.0,0.0</hprOffset>
                    <fov>59.0,46.0</fov>
                </chan>
            </pwin>
        </pipe>

       <!-- Configure right pipe -->
        <pipe>
            <screen>1</screen>
            <pwin>
                <!-- set pipe-window to fullscreen, no border -->
                <fullscreen>1</fullscreen>
                <border>0</border>
                <!-- Configure a single channel on pipe-window -->
                <chan>
```

```
                        <viewrange>0.64,1.0,0.0,1.0</viewrange>
                        <hprOffset>-53.333,0.0,0.0</hprOffset>
                        <fov>59.0,46.0</fov>
                    </chan>
                </pwin>
            </pipe>

           <!-- Configure left pipe -->
            <pipe>
                <screen>2</screen>
                <pwin>
                    <!-- set pipe-window to fullscreen, no border -->
                    <fullscreen>1</fullscreen>
                    <border>0</border>
                    <!-- Configure a single channel on pipe-window -->
                    <chan>
                        <viewrange>0.0,0.36,0.0,1.0</viewrange>
                        <hprOffset>53.333,0.0,0.0</hprOffset>
                        <fov>59.0,46.0</fov>
                    </chan>
                </pwin>
            </pipe>
        </display>

        <module>
            <!-- Loader module: loads models into world -->
            <class>pfvmLoader</class>
            <data>
                <Model>
                    <FileName>esprit.flt</FileName>
                </Model>
            </data>
        </module>

        <!-- Add a trackball navigation module -->
        <module>
            <class>pfvmTrackball</class>
        </module>

        <!-- Add a picking module -->
        <module>
            <class>pfvmPicker</class>
        </module>

    </viewer>
```

Module Scoping, Multiple Worlds and Multiple Views

In more complex pfvViewer applications, you can create multiple views and/or multiple worlds. Each view will always render (view) one of the specified worlds. Each world may be viewed by zero, one, or more views at any point during the life of the application.

Note: You can direct views from one world to another during the course of an application.

The following simple XML configuration file defines two worlds, each being rendered into a separate view:

```
<?xml version="1.0" ?>
<viewer>

    <!-- Specify two worlds, and assign each a unique name -->
    <world>
        <name>world0</name>
    </world>

    <world>
        <name>world1</name>
    </world>

    <!-- Specify two views, assign each a unique name,
         and direct to the corresponding world -->
    <view>
        <name>view0</name>
        <world>world0</world>
    </view>

    <view>
        <name>view1</name>
        <world>world1</world>
    </view>

    <!-- Specify two instances of the pfvmLoader module.
         By scoping these modules to different worlds,
         each loader module will add its geometry to the
         appropriate scene graph -->

    <module>
```

```
            <class>myLoader</class>
            <scope>world,world0</scope>
            <data>
                <Model>
                    <FileName>esprit.flt</FileName>
                </Model>
            </data>
        </module>

        <module>
            <class>myLoader</class>
            <scope>world,world1</scope>
            <data>
                <Model>
                    <FileName>truck.pfb</FileName>
                </Model>
            </data>
        </module>

        <!-- Specify two instances of the pfvmNavigator module.
             By scoping these modules to different views, each module
             will take care of navigation within the appropriate view -->

        <module>
            <class>myNavigator</class>
            <scope>view,view0</scope>
        </module>

        <module>
            <class>myNavigator</class>
            <scope>view,view1</scope>
        </module>

    </viewer>
```

You can achieve the same result through API calls, as shown in the following program:

```
#include <Performer/pfdu.h>
#include <Performer/pfv/pfvViewer.h>

main (int argc, char *argv[])
{
    pfvModule* module;

    // Initialize Performer.
    pfInit();

    // Initialize loading of model files.
    pfdInitConverter("flt");
    pfdInitConverter("pfb");

    // Create a new pfvViewer.
    pfvViewer* viewer = new pfvViewer;

    //Create first world.
    pfvWorld* w0 = viewer->createWorld();

    //Create second world.
    pfvWorld* w1 = viewer->createWorld();

    //Create first view. v0 becomes vieewer's current view.
    pfvView* v0 = (pfvView*)(viewer->createView());

    // Direct first view to first world.
    v0->setTargetWorld(w0);

    // Add navigation module to viewer's current view (v0).
    module = pfvModule::load("pfvmNavigator");
    viewer->addModule(module, PFV_SCOPE_VIEW);

    //Create second view. v1 becomes viewer's current view.
    pfvView* v1 = (pfvView*)(viewer->createView());

    // Direct second view to second world.
    v1->setTargetWorld(w1);

    // Add navigation module to viewer's current view (v1).
    module = pfvModule::load("pfvmNavigator");
    viewer->addModule(module, PFV_SCOPE_VIEW);
```

```
    // Configure/Initialize pfvViewer
    viewer->config();

    // Add car model to first world
    w0->addNode(pfdLoadFile("esprit.flt"));

    // Add truck model to first world
    w1->addNode(pfdLoadFile("truck.pfb"));

    // Start viewing
    viewer->run();
}
```

Extending a pfvViewer—Writing Custom Modules

A pfvViewer accepts user-written modules and incorporates their functions into its behavior. In order to extend a pfvViewer, you can write a new module. The following very simple module informs the pfvViewer to invoke the **handleEvent()** method and to print a message when the `F1` key is pressed:

```
class myModule : public pfvModule
{
public:

    myModule::myModule()
    {
        char keys[64];

        // Declare what keyboard inputs this module is interested in.
        sprintf(keys,"%c",PFVKEY_F1);
        bindKeys(keys);
    }

    myModule::~myModule() {;}

    // Keyboard event handler. pfvViewer calls this method every
    // time the user hits the F1 key.
    int handleEvent(int evType, char key)
    {
        printf("myModule::handleEvent called for key %s\n",
               pfvInputMngr::getKeyName(key) );
        return 0;
```

```
    }
};
```

In order to add this module to a pfvViewer program, add the following line:

```
viewer->addModule(new myModule);
```

Note that if this module is scoped to a view, pfvViewer will only inform the module of `F1` key presses within channels belonging to that specific view. Similarly, if this module is scoped to a specific world, pfvViewer would inform the module of `F1` key presses over any channel belonging to any view currently viewing such world.

You can scope this module to a world by making the following call:

```
pfvWorld* w;
viewer->addModule(new myModule, PFV_SCOPE_WORLD);
```

You can scope this module to a view by making the following call:

```
pfvView* v;
viewer->addModule(new myModule, PFV_SCOPE_VIEW);
```

The following example illustrates how to implement a basic custom module that controls the camera position (for the first view in pfvViewer's list) based on the mouse position:

```
class myModule : public pfvModule
{
public:
    myModule(){
        bindCallback(PFV_CB_FRAME);
    }

    ~myModule(){;}

    void frame() {
        // Only set eye for view0 if mouse is over view0
        if(pfvInputMngr::getFocusViewIndex()!=0)
            return;

        pfVec3 xyz,hpr;
        // Get current eye position (we don't want to change xyz)
        viewer->getView(0)->getEye(xyz,hpr);

        float mx, my;
```

```
        // Get current mouse position in view-normalized values
         // (0.0 to 1.0)
         pfvInputMngr::getViewNormXY( &mx, &my );

        // Compute new values for Heading and Pitch based on mouse
         // position
         hpr[0]= (mx-0.5f)*180.0f;
         hpr[1]= (my-0.5f)*-90.0f;

         // Set new eye position for view0
         viewer->getView(0)->setEye(xyz,hpr);
    }
};
```

Extending a pfvViewer—Module Entry Points

A module can gain program control at the various stages of rendering. The following are some of these stages:

- Event-Driven methods

  handleEvent()

  Called in response to a key-press event. A pfvViewer invokes this method only if the pressed key was bound by this module and if the event was generated over a view relevant to the module.

- Configuration methods (called once in the life of the application)
  - **preConfig()**

    Called before pfvViewer calls **pfConfig()**.
  - **postConfig()**

    Called after the pfvViewer calls **pfConfig()**.
- Run-Time methods (called every frame):
  - **sync()**

    Called each frame immediately after pfvViewer calls **pfSync()**.
  - **frame()**

    Called each frame after pfvViewer calls **pfFrame()**.

  - **preCull()**

    Called in all CULL processes before calling **pfCull()**.

  - **postCull()**

    Called in all CULL processes after calling **pfCull().**

  - **preDraw()**

    Called in all DRAW processes before calling **pfDraw()**.

  - **postDraw()**

    Called in all DRAW processes after calling **pfDraw()**.

  - **overlay()**

    Called in all DRAW processes after **postDraw()** callbacks.

- Enter and exit methods (called for scoped modules only)

  - **enterWorld()**

    A pfvViewer invokes this method on view-scoped modules to inform them that their view is about to start viewing a new world.

  - **exitWorld()**

    A pfvViewer invokes this method on view-scoped modules to inform them that their view is about to stop viewing current world.

  - **enterView()**

    A pfvViewer invokes this method on world-scoped modules to inform them that a new view is about to start viewing their world.

  - **exitView()**

    A pfvViewer invokes this method on world-scoped modules to inform them that a view is about to stop viewing their world.

Picking, Selection, and Interaction

The library `libpfv` also provides a framework for specifying custom interaction behavior through the pfvPicker, pfvInteractor, and pfvSelector classes.

A single pfvPicker instance will be able to coordinate multiple interaction classes derived from pfvInteractor and/or pfvSelector.

The following example illustrates how to derive entities from the pfvInteractor class in order to be able to highlight the geometry under the mouse cursor:

```
#include <Performer/pfv/pfvViewer.h>
#include <Performer/pfv/pfvInputMngrPicker.h>
#include <Performer/pr/pfHighlight.h>
#include <Performer/pf/pfLightSource.h>
#include <Performer/pfutil.h>
#include <Performer/pfdu.h>

class myInteractor : public pfvInteractor
{
public:

    myInteractor(){
        // create and configure a pfHighlight instance
        hl = new pfHighlight;
        hl->setMode( PFHL_LINES );
        hl->setColor( PFHL_FGCOLOR, 1.0f, 1.0f, 0.0f );
    }
    ~myInteractor(){ pfDelete(hl); }

    // startHlite will be called by picker once whenever mouse cursor
    // is moved over some geometry after being pointed away from all
    // geometry.
    int startHlite( pfvPicker*p, int prmsn ){
        // Obtain a pointer to the pfNode that was picked by picker
        p->getPickResults(&node);
        // Traverse picked node and highlight it
        pfuTravNodeHlight( node, hl );
        // return 1 indicating we accept highlighted state
        return 1;
    }

    // updateHlite will be called by picker on each frame as long as
    // mouse cursor remains over some geometry.
    int updateHlite( pfvPicker* p,int ev,int prmsn, pfvPickerRequest*r
){
        pfNode* curnode = node;
        // Obtain a pointer to the pfNode that was picked by picker
        p->getPickResults(&node);
```

```
        // if node picked by picker is not the node that is
        //  currently highlighted
        if(node!=curnode)
        {
            // De-highlight previously highlighted node
            pfuTravNodeHlight( node, hl );
            // Traverse picked node and highlight it
            pfuTravNodeHlight( curnode, NULL );
        }
        return 1;
    }

    // endHlite will be called by picker once whenever mouse cursor is
    // moved away from all geometry..
    void endHlite( pfvPicker* p ){
        // De-highlight previously highlighted node
        pfuTravNodeHlight( node, NULL );
    }

private:

    pfHighlight* hl;
    pfNode *node;
};


class myModule : public pfvModule
{
public:

    myModule(){
        // This module will use two callbacks (entry-points):
        bindCallback(PFV_CB_POSTCONFIG);
        bindCallback(PFV_CB_FRAME);
    }

    ~myModule(){;}

    // postConfig is called once by pfvViewer, after
    // calling pfConfig().
    void postConfig(){
        // Create a pfvInputMngrPicker instance
        picker = new pfvInputMngrPicker;
        // Set up picker so it will automatically isect scene and
        // allow interactors to highlight
```

```
        picker->setState(PFPICKER_ALLOW_HLITE, NULL, NULL);
        // Create an instance of our custom interactor
        ia = new myInteractor;
        // set up a pointer to our interactor on scene's root-node
        ia->nodeSetup( viewer->getWorld(0)->getScene(), picker);
    }

    // on every frame, call picker->update() from APP process.
    void frame() {
        picker->update();
    }

private:
    pfvInputMngrPicker* picker;
    myInteractor* ia;
};

int
main (int argc, char *argv[])
{
    pfInit();

    pfvViewer* viewer = new pfvViewer();

    pfFilePath(".:/usr/share/Performer/data");

    pfdInitConverter("esprit.flt");

    viewer->addModule(new myModule);
    viewer->addModule(pfvModule::load("pfvmTrackball"));

    viewer->config();

    viewer->addNode(new pfLightSource);
    viewer->addNode(pfdLoadFile("esprit.flt"));

    viewer->run();
}
```

More Sample Programs, Configuration Files, and Source Code

OpenGL Performer provides many `libpfv` sample programs, configuration files, and source code for modules in the following directories:

- `/usr/share/Performer/src/pguide/libpfv/picker` (IRIX and Linux)
  `%PFROOT%\Src/pguide\libpfv\picker` (Microsoft Windows)

  The samples in this directory demonstrate the use of pfvPicker and derived classes as well as how to extend the pfvInteractor and pfvSelector classes to implement your custom interaction behaviors.

- `/usr/share/Performer/src/pguide/libpfv/viewer` (IRIX and Linux)
  `%PFROOT%\Src\pguide\libpfv\viewer` (Microsoft Windows)

  The samples in this directory demonstrate how to do the following:

  - Load models into pfvViewer applications.
  - Load standard or custom modules.
  - Create pfvViewers with multiple camera positions (views).
  - Create pfvViewers with multiple independent scene graphs (worlds).
  - Create complex multichannel display configurations.
  - Load complex display configurations from an XML file.
  - Write custom modules.
  - Compile modules into re-usable DSOs.

- `/usr/share/Performer/src/pguide/libpfv/viewer/modules`
  (IRIX and Linux)
  `%PFROOT%\Src\pguide\libpfv\viewer\modules` (Microsoft Windows)

  This directory contains source code for the following modules:

  - `pfvmDrawStyle`
  - `pfvmEarthSky`
  - `pfvmLoader`
  - `pfvmLogo`
  - `pfvmNavigator`
  - `pfvmPicker`

  - `pfvmSnapshot`
  - `pfvmStats`
  - `pfvmTrackball`
  - `pfvmWorldSwitcher`

- `/usr/share/Performer/config` (IRIX and Linux)
  `%PFROOT%\Config` (Microsoft Windows)

  This directory contains examples of pfvViewer configuration files, denoted by the `.pfv` filename extension.

Image Gallery

This appendix contains views of some of the models that come with OpenGL Performer. The first nine 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 C-1 Simulated View of an Atrium

The image in Figure C-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 OpenGL Performer software distribution.

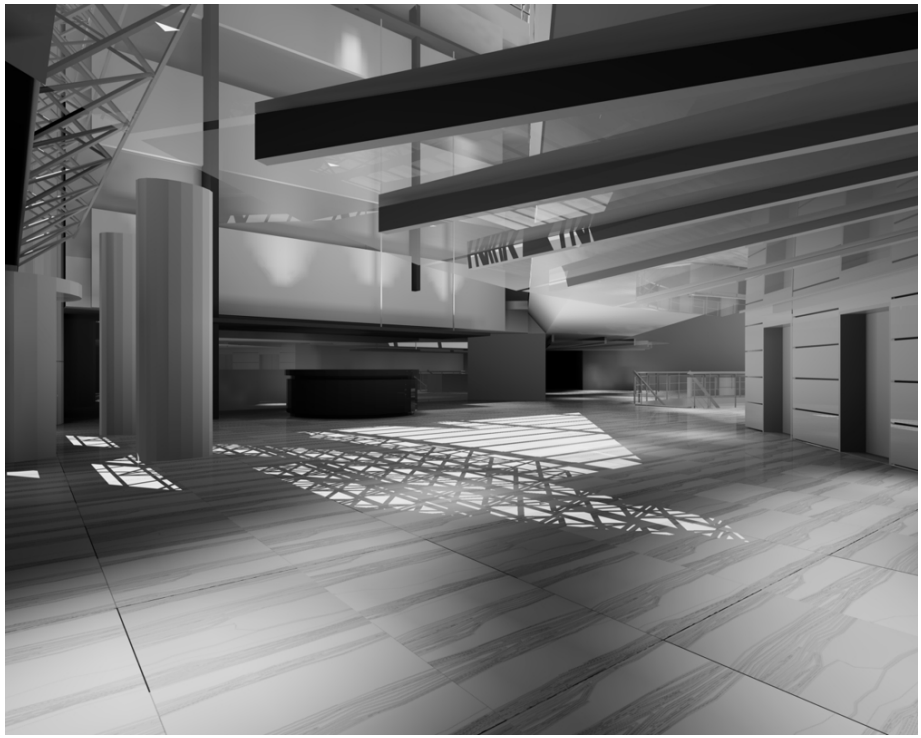

Figure C-2 Another Simulated View of the Atrium

The image in Figure C-2 was also created by A.J. Diamond, Donald Schmitt and Company, from the same database.

Figure C-3 Simulated View of a Castle

The image in Figure C-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 C-4 Simulated Hallway View

The image in Figure C-4 was created by A.J. Diamond, Donald Schmitt and Company.

Figure C-5 Simulated Hotel Lobby

The image in Figure C-5 was created by Design Vision Inc., Toronto. For more information, call 416-585-2020.

Figure C-6 Simulated Waiting Room

The image in Figure C-6 was created by Digital Architecture, Isao Nagaoka, and Joe Henke, New York. For information, call 212-587-4148.

Figure C-7 Simulated Conference Room

The image in Figure C-7 was created by Advanced Graphics Applications.

Figure C-8 Parliament Stairway

The image in Figure C-8 was created by A.J. Diamond, Donald Schmitt and Company.

Figure C-9 Unity Temple Interior

The image in Figure C-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 OpenGL Performer software distribution.

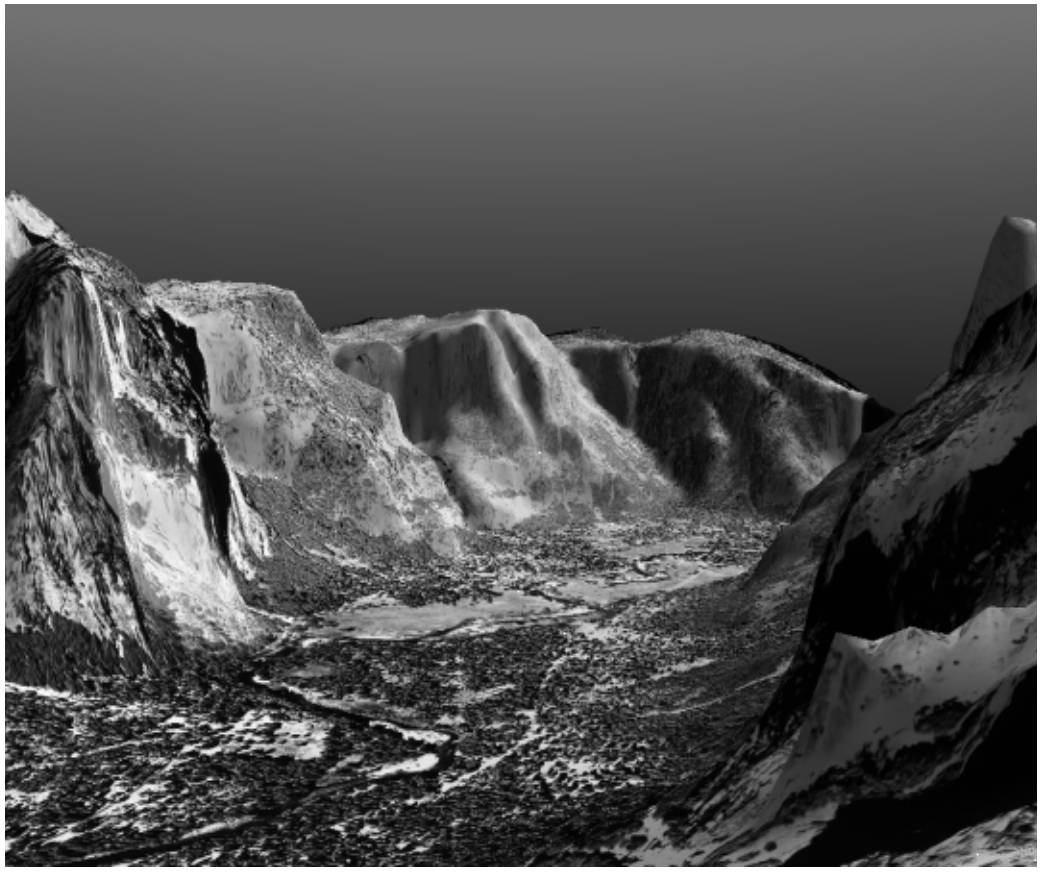

Figure C-10 Yosemite

The image in Figure C-10 was created by Delphi International with Yosemite image data courtesy of the National Park Service. This image used ClipTexture to drape 0.5m image data onto an Active Surface Definition terrain with 5m elevation data. Delphi International is a global provider of high-end data visualization software and services. Image copyright of Delphi International.

Figure C-11 DI-Guy

The image in Figure C-11 was created with DI-Guy, realistic humans for virtual environments, produced by Boston Dynamics Inc., using run-time morphing for smoothly animated and interactive figures. Image copyright of Boston Dynamics Inc.

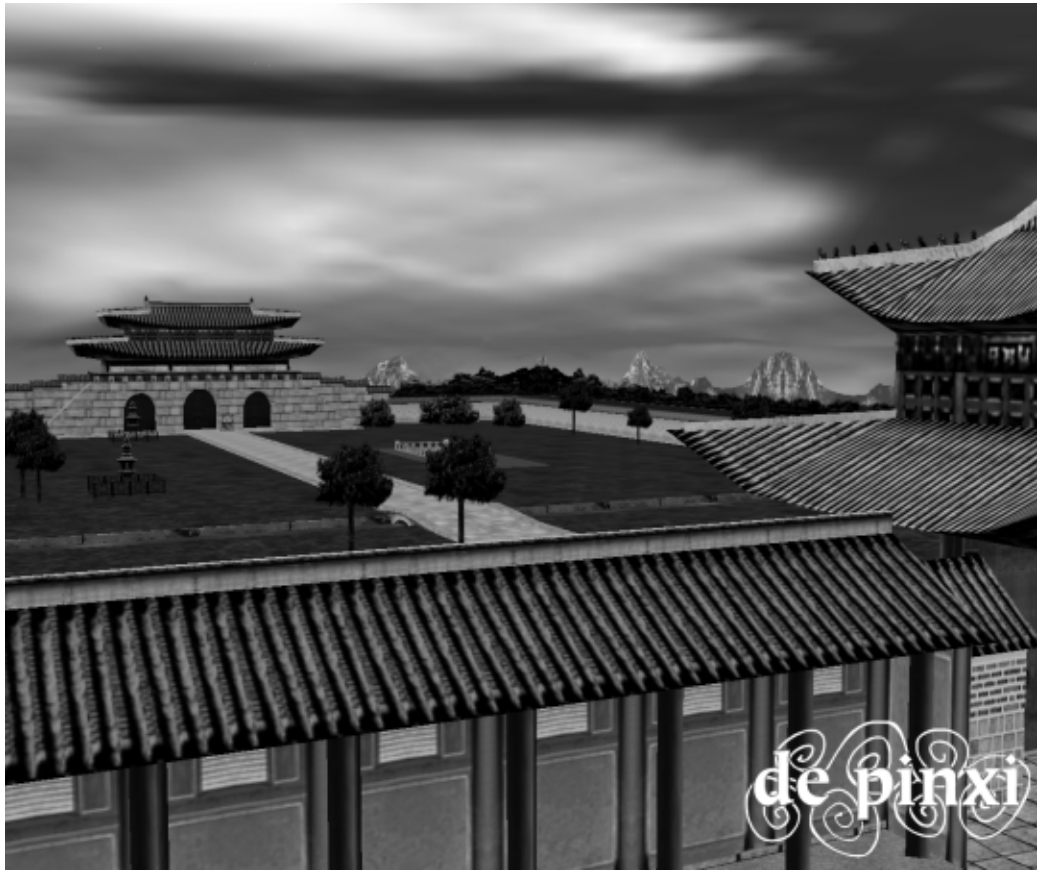

Figure C-12 Palace

The image in Figure C-12 is a screen snapshot of a 60Hz fly through of an archeological reconstruction of the Palace from Chosun Dinasty in Seoul, Korea. de Pinxi uses Performer for its interactive experiences. The model was created from ancient maps by de Pinxi for LG, Korea. Image copyright of de Pinxi.

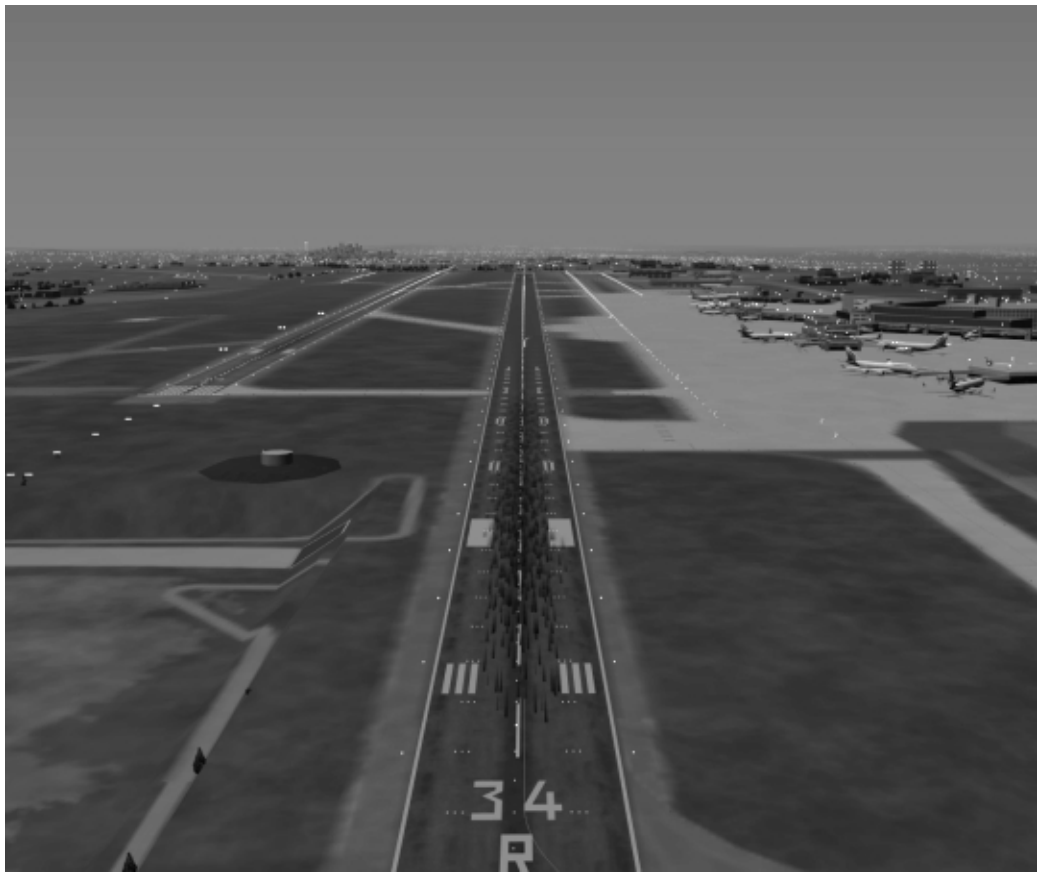

Figure C-13 Seattle-Tacoma International Airport

IVEX provides real-time visual systems for commercial and military flight simulation. Superior integrated image generators, database scenarios, and complete system integration services use clipmapping, ASD, and calligraphic lights. The image in Figure C-13 is provided courtesy of IVEX.

Figure C-14 Hasparen

Thomson Training & Simulation is the leading simulator manufacturer outside of North America. The image in Figure C-14 is provided courtesy of Thomson Training & Simulation.

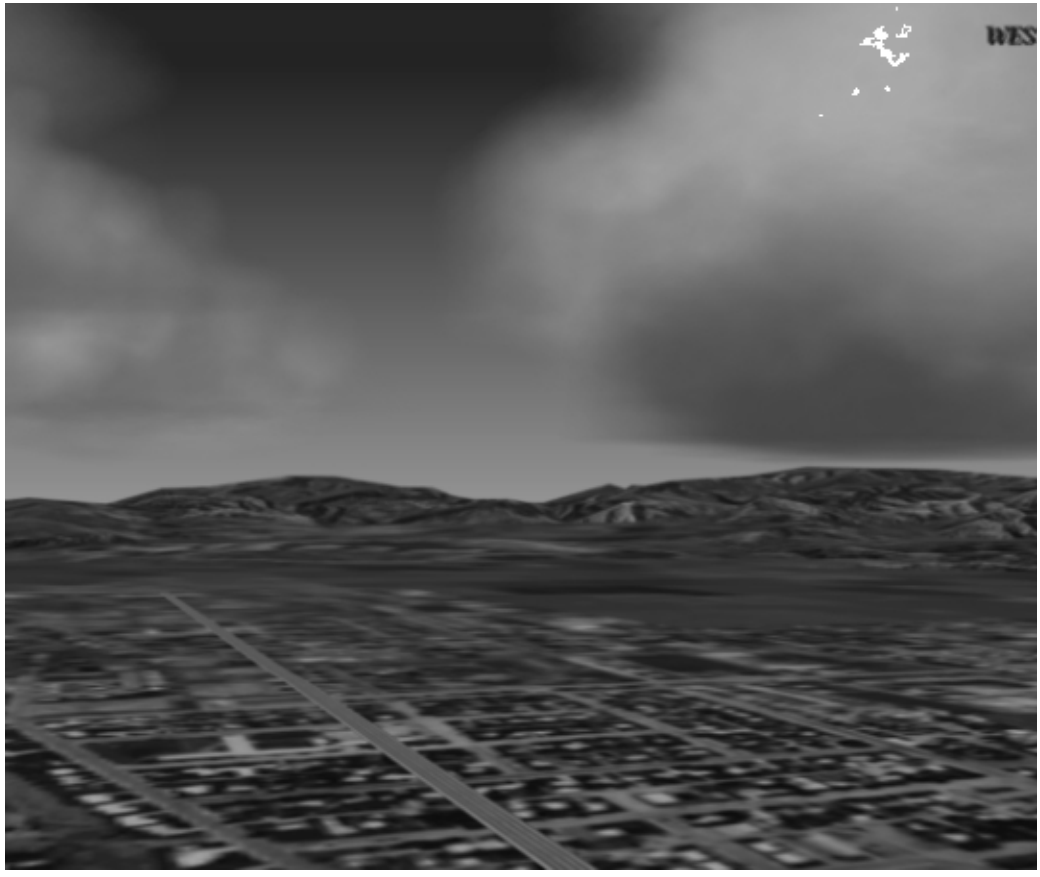

Figure C-15 Clouds

Southwest Research Institute received an R&D 100 award for development of the patented Weather Environment Simulation Technology (WEST) process. WEST enables volumetric rendering of real-world weather conditions through the use of dynamically shaped billboards. The process assures real-time performance by prioritizing weather elements in the field of view. The image in Figure C-15 is a copyright of SWRI.

Figure C-16 Ocean and Marine Effects Simulation

Vega Marine is OpenGL Performer-based visual simulation software for the development of maritime applications. This module, one of many within the Vega Development system, generates a real-time dynamic ocean modeled as a textured surface with wave heights and periods corresponding to sea states. Vega Marine provides special effects essential to realistic maritime simulation. These effects include wakes, wind effects on water, constant tension and constant length lines, moored buoys, depth (bathymetry) effects, foam, flotsam, surf, horizon glow, and glare from the sun. The image in Figure C-16 is a copyright of Paradigm Simulation Inc.

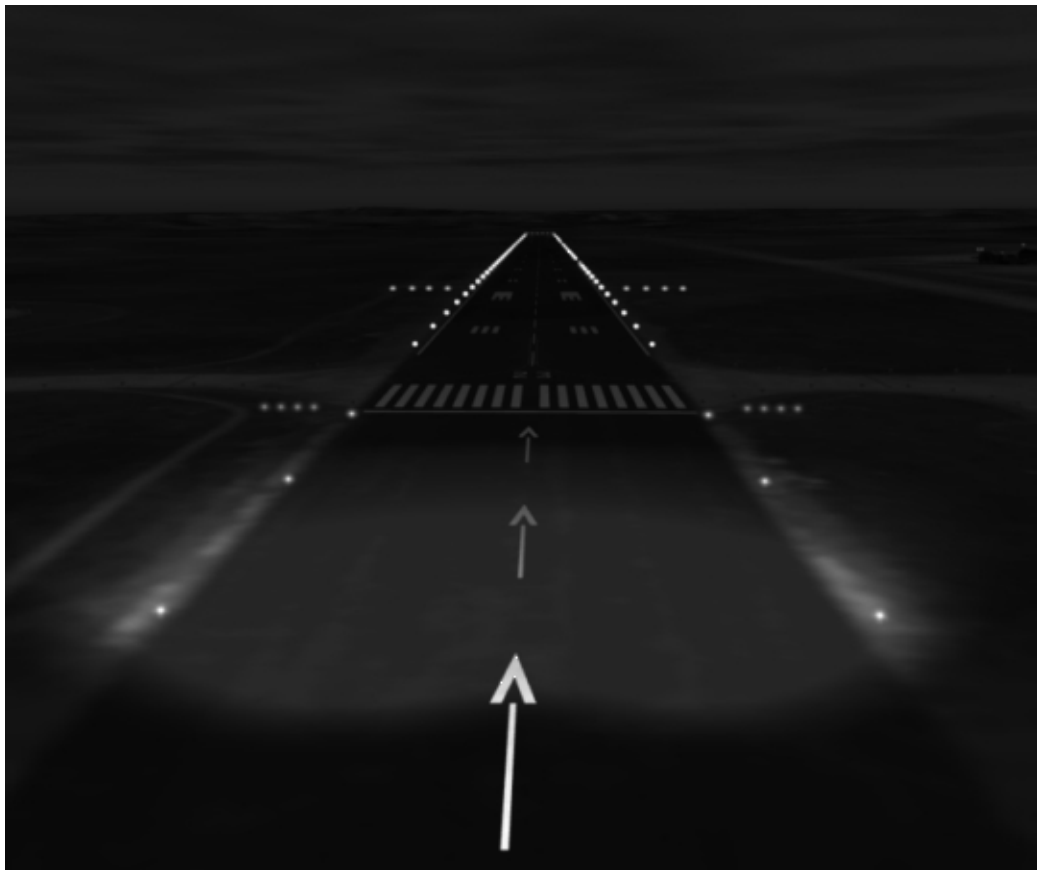

Figure C-17 Night Image

The image in Figure C-17 shows an aircraft 60 Hz visual system with advanced visual effects, including runway aeronautical model lighting. Landing lobes implementation affects terrain following and cultural features illumination. The simulation uses fully geospecific real-time database paging and advanced weather model. The image is provided courtesy of Construcciones Aeronauticas S.A., Madrid, Spain.

Glossary

arena

A portion of memory shared by OpenGL Performer processes.

billboard

A slice of a geometry that rotates with the viewer so that the entire geometry appears rendered, even though only a slice of it has been rendered.

channel

A view of objects in a scene, based on the location and orientation of the camera in the scene viewing frustum.

cull

Eliminates all geometries out of view from rendering.

clip texture

Virtualizes MIPmapped textures using hardware and software support, so that only the texels in the region close to the viewer (known as the clipped region) need to be loaded in texture memory. (Also known as ClipMap.)

dynamic shared object (DSO)

An object that can be shared between applications at run time.

dynamic link library (DLL)

The Microsoft Windows equivalent of a dynamic shared object (DSO).

frustum

Computer-generated objects can be projected into an artificial viewing area, called a frustum. A frustum is in the form of a truncated pyramid, shown in Figure 5-3, between the base of the viewing volume, called the far plane, and the near plane.

heap
A portion of memory reserved for graphics.

libpf
The pf Performer library; a higher level library that relies on libpr.

libpr
The pr Performer library containing basic Performer tools.

libpfdu
The du Performer library containing database utilities, helpful for loading scene graphs.

load
(1) Transferring from disk to memory. (2) The processing burden of rendering a frame, and can be defined as the rendering time divided by the desired frame period.

model view
Same as object space.

node
A class derived from pfNode. A node can be part of a scene graph.

object space
A coordinate system in a subsection of a scene graph. Also referred to as *model view.*

Perfly
The demonstration program distributed with OpenGL Performer.

pipe
Renders the visual data, contained in the viewing frustum, to a window.

scene graph
A hierarchy of nodes. The hierarchy specifies the order in which the nodes are processed.

stress
Stress is directly related to the graphics load; the more complex the display, the higher the system stress.

texture mapping

Applies textures, such as the appearance of an orange, to the surface of a geometry.

traversal

An scene graph action, such as a draw action, going from one node to another.

viewing frustum

A truncated pyramid defined by the near and far clipping planes, and by the horizontal and vertical field of view. See Figure 4-3. Only those shapes in the frustum are visible to the viewer.

viewing volume

The viewing volume is the pyramid, shown in Figure 5-3, formed between the eyepoint and the vertical and horizontal field of view.

viewport

A viewport refers to each channel in a window.

world space

The coordinate system of the root node, in which all shapes in a scene graph can reside.

Index

C

D

E

F

G

H

I

L

M

N

O

P

R

U

V

W

X